Effizienzsteigerung bei Auslegung und Inbetriebnahme mechatronischer Systeme durch Verwendung modellbasierter Entwicklungsmethoden auf Basis offener Standards

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Maschinenbau und Verfahrenstechnik der
Universität Duisburg-Essen
zur Erlangung des akademischen Grades

eines

Doktors der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

von

Nils Menager
aus
Hamburg

Gutachter: Prof. Dr.-Ing. Dieter Schramm
Prof. Dr.-Ing. Markus Bröcker
Prof. Dr.-Ing. Frank Lobeck
Tag der mündlichen Prüfung: 20.06.2017

Danksagung

Die vorliegende Arbeit entstand während meiner Zeit bei der Bosch Rexroth AG in Lohr a. Main. Mein besonderer Dank gilt Herrn Prof. Dieter Schramm für die Betreuung und die Unterstützung während der Promotion. Ebenfalls danken möchte ich Herrn Prof. Markus Bröcker und Herrn Prof. Frank Lobeck für die Übernahme des Korreferats und die kritische Durchsicht der Arbeit.

Weiterhin danken möchte ich den zahlreichen Kollegen bei Bosch Rexroth, die mir stets mit Rat und Tat zur Seite standen. Allen voran gilt mein Dank Herrn Dr.-Ing. Lars Mikelsons, der mich jederzeit unterstützt hat und mit zahlreichen fachlichen Diskussionen einen wertvollen Beitrag zu der Arbeit geleistet hat. Darüber hinaus möchte ich Niklas Worschech und Andreas Hofmann danken, die ebenfalls jederzeit für konstruktive Fachdiskussionen zur Verfügung standen. Herrn Dr.-Ing. Tobias Wittkopp möchte ich insbesondere für die Möglichkeit der Promotion bei Bosch Rexroth danken. Darüber hinaus danke ich allen Studenten, deren Ergebnisse zur Entstehung dieser Arbeit beigetragen haben. Dies gilt insbesondere für die Arbeiten von Rüdiger Kampfmann.

Abschließen möchte ich mit einem ganz besonderen Dank für das Verständnis und die Unterstützung meiner Familie.

Kurzzusammenfassung

Die Time-to-Market eines neuen mechatronischen Systems hat einen entscheidenden Einfluss auf den wirtschaftlichen Erfolg des Produktes. Eine Reduzierung der Time-to-Market muss daher eines der Hauptziele eines Unternehmens sein. Während die Produkte durch die Entwicklung hin zu cyber-physischen Systemen im Kontext von Industrie 4.0 zunehmend komplexer werden, sind die zugrundeliegenden Produktentwicklungsprozesse noch immer unverändert. Eine Betrachtung des in der VDI Richtlinie 2206 definierten klassischen Produktentwicklungsprozesses für mechatronische Systeme zeigt einige Potentiale zur Effizienzsteigerung auf. In sämtlichen Phasen der Produktentwicklung, von der Definition der Anforderungen bis hin zum Betrieb, ist es möglich, durch eine konsequente Umsetzung eines durchgängigen, modellbasierten Engineerings Zeiteinsparungen zu erzielen. In der vorliegenden Arbeit wird vor allem die Maschinenbau- und Automatisierungsbranche betrachtet und ein modellbasierter Entwicklungsprozess vorgestellt, der eine Effizienzsteigerung hauptsächlich innerhalb der Phasen der Auslegung und der Inbetriebnahme ermöglicht. Ein durchgängiges, modellbasiertes Engineering lässt sich jedoch nicht auf Basis kommerzieller Tools umsetzen. Der Fokus liegt auf einer Lösung, die auf offenen Standards basiert und somit toolunabhängig ist. An zwei praxisrelevanten Beispielen wird dargestellt, wie die Methoden in der Industrie zu einer Verbesserung des Produktentwicklungsprozesses beitragen können. Dazu wird in einem ersten Beispiel ein Gleichlaufsystem für zwei hydraulische Achsen, ein Standardfall in der hydraulischen Antriebstechnik, modellbasiert entwickelt. Abschließend wird die modellbasierte Entwicklungsmethodik an einem zweiten Beispiel, einer komplexen Flaschenabfüllanlage, gezeigt. An diesem Beispiel wird verdeutlicht, wie sich die im Rahmen dieser Arbeit entwickelten Bausteine auf Basis offener Standards ebenfalls in kommerzielle Softwaretools integrieren lassen.

Abstract

The time-to-market of a new mechatronic system has a decisive influence on the economic success of the product. The reduction of the time-to-market must therefore be one of the main goals of each company. While the products are getting more and more complex, e.g. due to the increasing trend towards cyber-physical systems in the context of Industry 4.0, the underlying product development processes are still unchanged. On closer inspection, the classical product development process for mechatronic systems defined inside the VDI 2206 guideline shows up great potential for increasing efficiency. In all phases during the product development, from the design phase to the operation of the machine, it is possible to realize time savings by a consequent usage of model-based engineering methods. In this thesis, mainly the mechanical engineering and automation industry is considered and a model-based development process is presented, which allows an efficiency increase mainly within the phases of design and commissioning. However, an integrated model-based engineering cannot be realized on the basis of commercial tools. The focus is on a solution which is based on open standards and is thus tool-independent. It is shown in two practical examples how the methods contribute to improve the product development process. For this purpose, in a first example, a synchronized motion for two hydraulic axes, which represents a standard case in the hydraulic drive technology, is developed in a model-based way. Finally, the model-based development methodology is shown in a second example, a complex bottle filling machine. This example illustrates how the solutions based on open standards developed in this work can also be integrated into commercial software tools.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Dieses einleitende Kapitel beschreibt zunächst die Motivation für diese Arbeit. Anschließend wird der Stand der Technik dargelegt und die Arbeit in den wissenschaftlichen Kontext eingeordnet. Abschließend wird auf den Aufbau und die Zielsetzung der Arbeit eingegangen.

1.1 Motivation

Der wirtschaftliche Erfolg eines neuen Produktes hängt wesentlich von der Dauer zwischen der Konzeption und der Serienreife ab. Eine Reduzierung der Produktentwicklungszeit und somit der *Time-to-Market* muss für Unternehmen daher eine wesentliche Zielsetzung sein. Gleichzeitig werden die zu entwickelnden Anlagen im Maschinenbau und der Automatisierungsindustrie aufgrund stetig steigender Anforderungen zunehmend komplexer. Dies geht aus einer im Jahr 2013 durchgeführten Studie hervor, die die Produkte von morgen als intelligente und vernetzte Systeme beschreibt, die gleichzeitig über einen hohen Funktionsumfang und eine hohe Bedienerfreundlichkeit verfügen (vgl. Abbildung 1.1) [Gausemeier *et al.* (2013)].

Derartige Systeme erfordern ein reibungsloses Zusammenwirken von Komponenten vieler unterschiedlicher Domänen. In typischen mechatronischen Systemen befinden sich häufig mechanische, elektronische und informationstechnische Komponenten, jedoch können auch beispielsweise hydraulische Komponenten enthalten sein. Um die Potentiale mechatronischer Systeme vollständig nutzen zu können, ist es notwendig, von den bisherigen Entwicklungsprozessen für den Maschinenbau abzurücken. Bisher ist es die Regel, dass die Anlage vornehmlich von einer Domäne, dem Maschinenbau, entwickelt wird, und die elektronischen und informationstechnischen

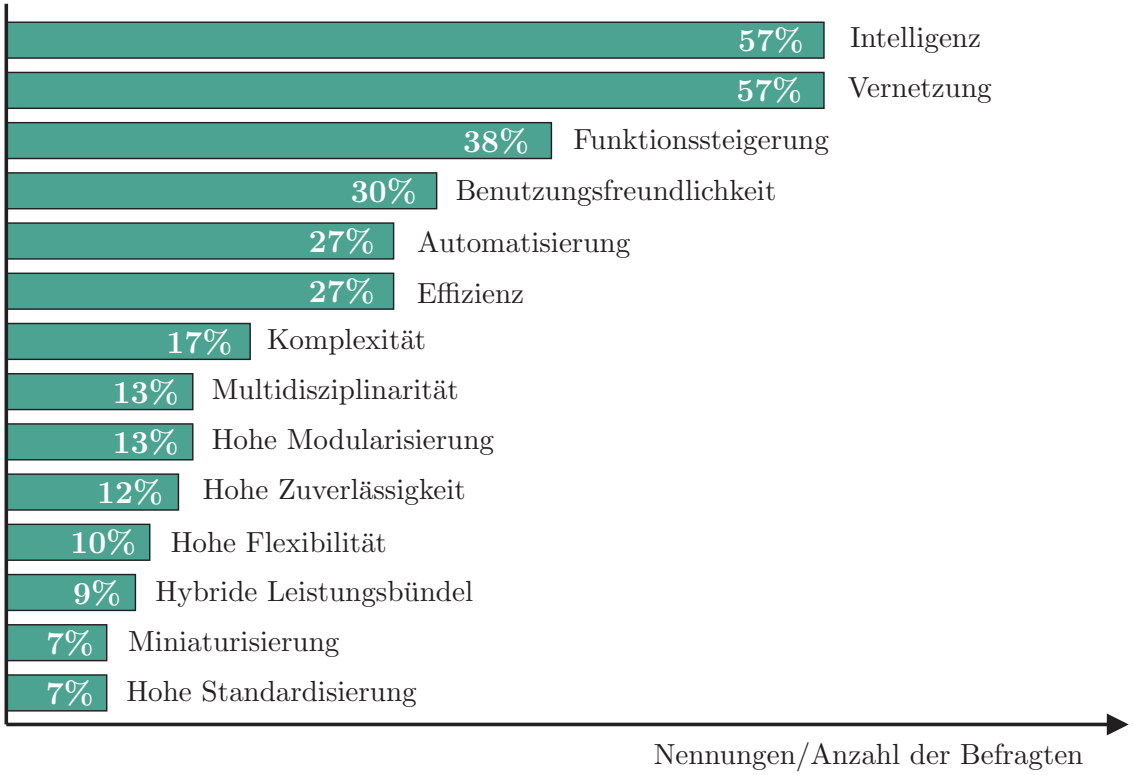

Abbildung 1.1: Eigenschaften der Produkte von morgen [Gausemeier *et al.* (2013)]

Komponenten in einem nachgelagerten Schritt integriert werden. Eine der größten Herausforderungen bei der Entwicklung mechatronischer Systeme besteht darin, die Potentiale der unterschiedlichen Domänen gleichberechtigt zu nutzen und diese im Sinne eines *Simultaneous Engineering* zum Optimum zu führen [Gehrke (2005)].

Ein erster Ansatz für die Entwicklung mechatronischer Systeme und Anlagen ist in der VDI Richtlinie 2206 des *Vereins Deutscher Ingenieure* [VDI2206 (2004)] enthalten. Diese enthält das aus der Softwareentwicklung bekannte V-Modell, welches auf die Anforderungen der Mechatronik angepasst wurde. Abbildung 1.2 zeigt das V-Modell der mechatronischen Produktentwicklung. Der mechatronische Entwicklungsprozess unterscheidet drei Phasen, den Systementwurf, den domänenspezifischen Entwurf sowie die Systemintegration. Innerhalb des Systementwurfs werden die allgemeinen, domänenübergreifenden Entscheidungen getroffen. Erst wenn das Gesamtkonzept feststeht, wird mit der domänenspezifischen Entwicklung begonnen. Die dort entwickelten Lösungen werden im Rahmen der Systemintegration zu einem Gesamtsystem zusammengefügt. Um die Eignung des Gesamtsystems bezüglich der zu Beginn festgelegten Anforderungen überprüfen zu können, wird eine Eigenschaftsabsicherung durchgeführt. Sind die zu Beginn festgelegten Anforderungen nicht oder nur teilweise erfüllt, beginnt der Zyklus von Neuem und die Auslegung der domänenspezifischen Komponenten wird überarbeitet und verbes-

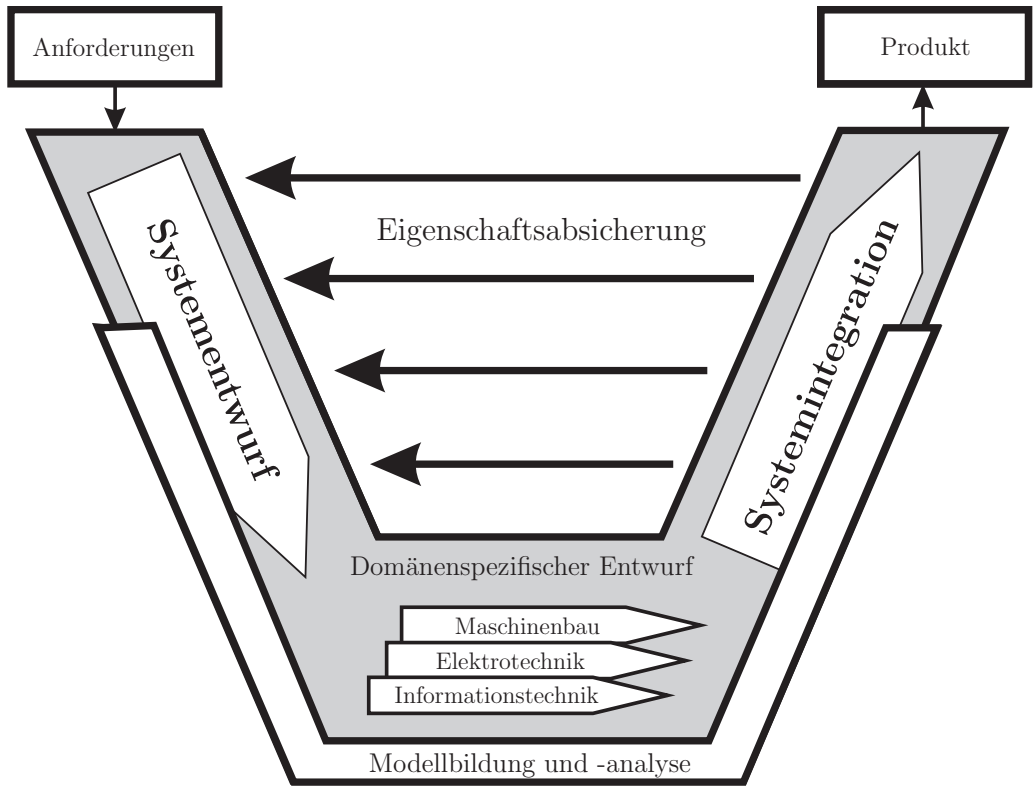

Abbildung 1.2: V-Modell der mechatronischen Produktentwicklung [VDI2206 (2004)]

sert. Dies geschieht solange, bis das Gesamtsystem die gegebenen Anforderungen erfüllt. Die Eigenschaftsabsicherung kann durch Simulationsmethoden unterstützt werden, sodass sich die Anzahl benötigter realer Prototypen in der Entwicklungsphase reduzieren lässt. Dieses Vorgehen ermöglicht kurze Iterationen innerhalb des Entwicklungsprozesses und führt damit zu Kosten- und Zeiteinsparungen.

Werden die bei der Entwicklung einer neuen Anlage auftretenden Phasen *Konzeption und Auslegung*, *Fertigung und Montage*, *Inbetriebnahme* und *Betrieb* betrachtet, macht die Phase der Inbetriebnahme mit bis zu 25% einen signifikanten Anteil an der Gesamtprojektdauer aus. Von diesen 25% entfallen etwa 90% auf die Inbetriebnahme der Elektrik sowie der Steuerungstechnik. Innerhalb der Inbetriebnahme der Steuerungstechnik wiederum werden 70% der Zeit für das Auffinden und die Beseitigung von Softwarefehlern benötigt [Wünsch (2008)]. Folglich beträgt der Anteil der Zeit, der für die Beseitigung der Softwarefehler benötigt wird, bis zu 15,75% der gesamten Projektlaufzeit. Abbildung 1.3 veranschaulicht den Sachverhalt graphisch.

Wird berücksichtigt, dass der Wertschöpfungsanteil der Software durch die zunehmende Vernetzung von Produkten in Zukunft weiter steigen wird [Kaufmann (2015)], wird die Inbetriebnahme der Anlage zukünftig zu einer Schlüsselphase im

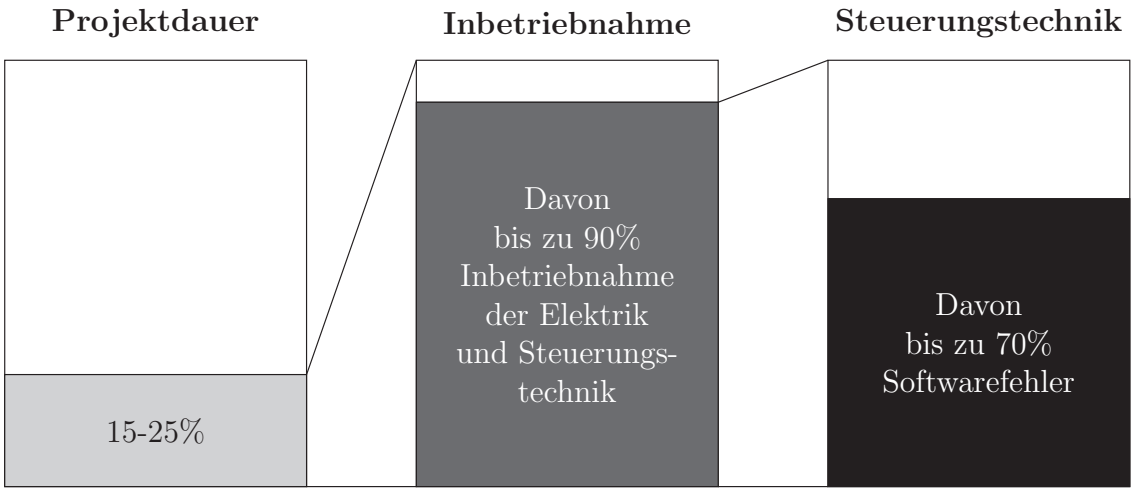

Abbildung 1.3: Zeitanteile bei der Inbetriebnahme [VDW-Bericht (1997)]

Rahmen der Produktentwicklung. Die wachsende Bedeutung der Software konnte ebenfalls in einer Trendstudie des VDMA (*Verband Deutscher Maschinen- und Anlagenbau*) aus dem Jahr 2015 nachgewiesen werden [VDMA (2015)].

Effizienzdefizite bei der Inbetriebnahme Bei genauerer Betrachtung weist die heutige Vorgehensweise bei der Inbetriebnahme in der Regel zwei gravierende Probleme auf. Für die Auslegung eines zu entwickelnden mechatronischen Systems wird in der Praxis, wie in der VDI Richtlinie 2206 empfohlen, mittlerweile immer häufiger auf Simulationsmethoden zurückgegriffen, wodurch die im V-Modell geforderte Eigenschaftsabsicherung auf schnelle und einfache Weise erfolgen kann. Dieses als *Virtual Prototyping* bezeichnete Vorgehen reduziert Zeit und Kosten, da einerseits die virtuellen Prototypen im Vergleich zu einem realen Prototypen deutlich schneller zu erstellen sind und andererseits keine Material- und Fertigungskosten anfallen. Gleichzeitig sind solche Modelle deutlich flexibler, da Änderungen und Anpassungen am System in späteren Entwicklungszyklen praktisch ohne weiteren Kostenaufwand umgesetzt werden können. Weiterhin kann der Prototyp an mehreren Standorten gleichzeitig verwendet werden. Es ist durch die Verwendung von virtuellen Prototypen problemlos möglich, Untersuchungen bei unterschiedlichen Umweltbedingungen durchzuführen, hierzu ist lediglich der entsprechende Parameter im Simulationsmodell zu ändern. Auch für die Simulation von gefährlichen Szenarien eignen sich virtuelle Prototypen.

Für eine Analyse des dynamischen Verhaltens des Modells ist es notwendig, zusätzlich einen Regelalgorithmus innerhalb der Simulationsumgebung zu implementieren, da das virtuelle System ohne einen Eingang in Form einer Stellgröße in der Regel keine Bewegungen durchführt. Beim Übergang von der Auslegungs- in die Inbetriebnahmephase werden die in der Auslegungsphase erstellten Modelle jedoch bisher nicht weiter verwendet. Stattdessen wird zu Beginn der Inbetriebnahmephase

komplett von vorne mit der Entwicklung und Implementierung eines Regelalgorithmus begonnen. Dieses Vorgehen hat wesentliche Nachteile. Neben dem zusätzlichen Zeit- und somit Kostenaufwand für eine Re-Implementierung des bereits vorhandenen Reglers stellt die Re-Implementierung stets eine potentielle Fehlerquelle dar. Die Implementierung von Reglercode während der Inbetriebnahme erfolgt darüber hinaus häufig in SPS-Programmiersprachen nach IEC 61131-3 [IEC (1993)] und somit in einer anderen Sprache als innerhalb der Simulationsumgebung. Es kann somit nicht gewährleistet werden, dass das Verhalten des Reglers identisch mit dem des Reglermodells aus der Simulationsumgebung ist, mit dessen Hilfe das System ausgelegt wurde. Ein weiteres Problem besteht in der Tatsache, dass mit der Inbetriebnahme und somit mit dem Testen der Software in der Regel erst angefangen werden kann, sobald die Anlage fertig gestellt und elektrisch vollständig angeschlossen wurde [Mewes (2005)]. Liegen in der Software Fehler vor, können diese erst in dieser Phase erkannt und beseitigt werden. Die nahezu fertige Anlage steht jedoch bereits bei dem Kunden und verursacht durch die Bindung von Kapital und Platz hohe Kosten. Das Ziel muss es also sein, bereits vor der Errichtung der Anlage die Software fertigzustellen und mit der Inbetriebnahme zu beginnen. Im Folgenden werden diese beiden Punkte eingehender beleuchtet.

Durch den Einsatz von *modellbasierten Entwicklungsmethoden* ist es möglich, die Effizienz der Systementwicklung deutlich zu steigern und somit die Time-to-Market zu reduzieren. Modellbasierte Entwicklung meint dabei die Konzipierung, Ausarbeitung sowie Verifikation und Validierung von komplexen Systemen mit einem interdisziplinären Systemmodell als gemeinsame Entwicklungsbasis, welches den gesamten Entwicklungsprozess begleitet. Eine Möglichkeit zur Reduzierung der Entwicklungszeit ist die Methode der *virtuellen Inbetriebnahme*, die durch die modellbasierte Entwicklung ermöglicht wird. Bei der virtuellen Inbetriebnahme kann die Inbetriebnahme bereits während der Auslegungsphase beginnen, da nicht auf die Fertigstellung der realen Anlage gewartet werden muss. Stattdessen wird die entwickelte Software an einem virtuellen Abbild der Anlage erprobt und optimiert. Die reale Inbetriebnahme besteht dann nur noch aus einer Feinoptimierung der Parameter in einer ansonsten vollständig getesteten Software. Die sich daraus ergebende Zeitersparnis lässt sich anhand von Abbildung 1.4 nachvollziehen. In dem oberen Teil der Abbildung ist der Projektverlauf ohne Verwendung der virtuellen Inbetriebnahme (VIBN) dargestellt. Es treten die klassischen Phasen Konstruktion, Fertigung/Montage und Inbetriebnahme auf, die zeitlich nacheinander durchlaufen werden. In diesem Fall wird mit der Inbetriebnahme erst begonnen, wenn die Anlage vollständig gefertigt und montiert wurde, was dem heutigen Regelfall entspricht. Der Projektverlauf bei Verwendung modellbasierter Entwicklungsmethoden ist in dem unteren Teil der Abbildung gezeigt. Parallel zur Konstruktion wird ein Simulationsmodell aufgebaut. Mit Hilfe dieses Modells kann bereits frühzeitig mit der virtuellen Inbetriebnahme begonnen werden. Nach Errichtung der Anlage benötigt die Feinoptimierung der Parameter innerhalb der Software lediglich einen Bruchteil der ursprünglichen Inbetriebnahmezeit. Die Notwendigkeit der Feinoptimierung re-

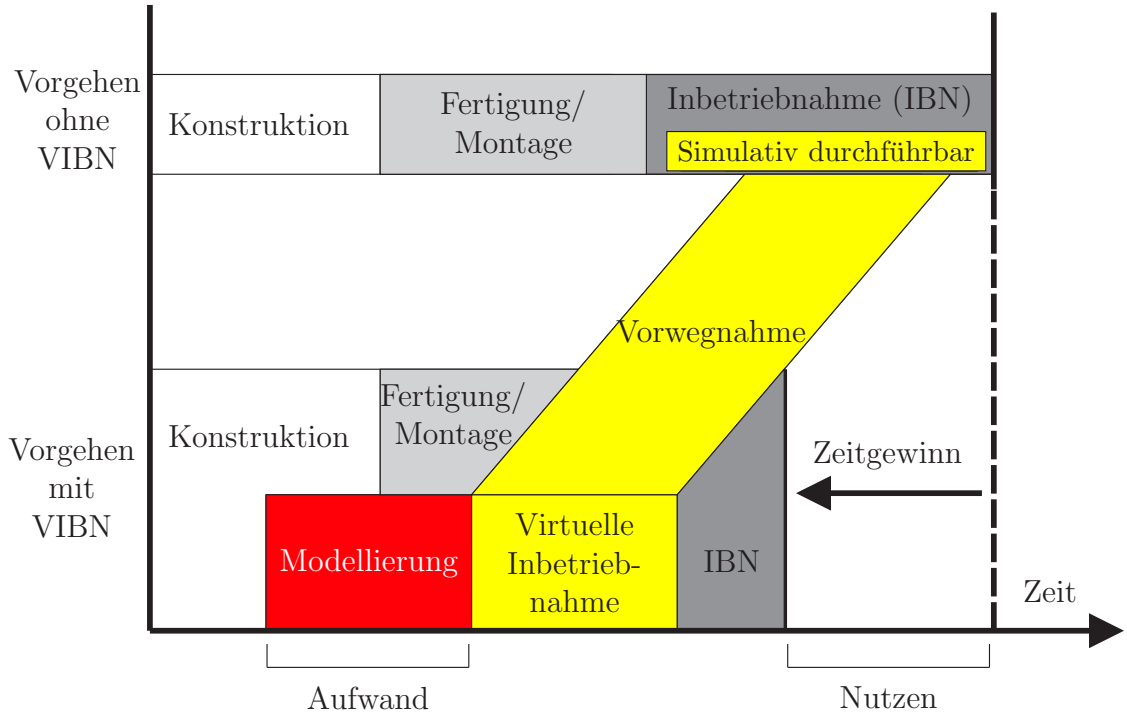

Abbildung 1.4: Grundidee der virtuellen Inbetriebnahme [Wünsch (2008)]

sultiert aus den Abweichungen zwischen Modell und realer Anlage. Gelingt es, die Modellgüte zu erhöhen, sodass das Verhalten des realen Systems durch das Modell sehr gut abgebildet wird, wird für die Feinoptimierung entsprechend wenig Zeit benötigt. Auf der anderen Seite benötigt in diesem Fall die Modellbildung deutlich mehr Zeit. An dieser Stelle muss zwischen dem Aufwand, den die Erstellung des Simulationsmodells verursacht, und dem Nutzen, der durch die kürzere Inbetriebnahmezeit an der realen Anlage entsteht, abgewogen werden. Insgesamt sorgt die Verwendung modellbasierter Entwicklungsmethoden dafür, dass die Phasen der Auslegung und der Inbetriebnahme nicht mehr strikt getrennt bearbeitet werden, sondern miteinander verschmelzen, wodurch sich die Zeitersparnis ergibt.

Neben der virtuellen Inbetriebnahme ist die *Codegenerierung* ein zweites wichtiges Instrument der modellbasierten Entwicklung. Ein Hauptziel der modellbasierten Entwicklung ist das durchgängige Engineering und die damit verbundene Wiederverwendung bereits vorhandenen Wissens. Wird berücksichtigt, dass in der Auslegungsphase bereits ein Regler innerhalb der Simulationsumgebung entwickelt wurde, um das dynamische Verhalten der Regelstrecke zu untersuchen, ist es naheliegend, dieses Wissen in die Inbetriebnahme zu transferieren, anstatt in dieser Phase wieder von vorne zu beginnen und das vorhandene Wissen zu verwerfen, indem der Regler erneut implementiert wird. Gelingt es, aus dem Simulationsmodell des Reglers ausführbaren Code zu generieren und diesen auf der realen Steuerungshardware auszuführen, kann auf eine Re-Implementierung verzichtet werden. Diese

Form der modellbasierten Reglerentwicklung wird auch als *Rapid Control Prototyping* bezeichnet. Dies stellt jedoch nur eine mögliche Anwendung der modellbasierten Entwicklung dar. Es ist ebenso denkbar, statt des Modells des Reglers das Modell der Regelstrecke über die Auslegungsphase hinaus weiterzuverwenden. Einen Anwendungsfall stellt die *modellbasierte Diagnose* dar. Parallel zum Betrieb der Anlage wird das Verhalten des Systems anhand eines Modells vorausgesagt. Das simulierte Verhalten wird mit dem tatsächlich gemessenen Verhalten der realen Anlage verglichen. Treten Abweichungen auf, lässt dies auf einen Fehlerfall der Anlage schließen. Werden zusätzlich spezielle Fehlermodelle in die Komponentenmodelle integriert, lässt sich unter Umständen sogar der exakte Fehlerfall ermitteln [Mikelsons & Su (2014)] [Leweling (1995)]. Die für eine modellbasierte Diagnose notwendigen Komponentenmodelle sind jedoch sehr komplex, was eine Ausführung auf aktuellen Steuergeräten erschwert. An dieser Stelle kann stattdessen eine *heuristische Diagnose* eingesetzt werden, die auf Erfahrungswissen von Experten beruht. Sofern dieses Wissen in der Form *„Wenn Symptom s beobachtet wird, liegt Fehler d vor“* explizit vorliegt, kann dieses Wissen in Software verfügbar gemacht werden. Auf diese Weise kann auf die Integration komplexer Fehlermodelle verzichtet werden und die Diagnose mit Hilfe dieser Software erfolgen [Stein & Husemeyer (2001)]. Bei Verwendung derartiger Diagnosemodelle kann jedoch keine vollständige Durchgängigkeit gewährleistet werden, da diese Modelle systemspezifisch sind und zusätzlich zu den bestehenden Anlagenmodellen entwickelt werden müssen, wohingegen für die modellbasierte Diagnose die bereits bestehenden Modelle verwendet werden können.

Darüber hinaus können Streckenmodelle auch direkt für die Regelung verwendet werden. Neben klassischen Methoden der Regelung, die den Entwurf von Reglerstrukturen wie beispielsweise eines PID-Reglers verfolgen, kann die Regelung ohne zusätzliche Entwicklung eines Regelalgorithmus unmittelbar mit dem Streckenmodell erfolgen. Der einfachste Fall ist die Verwendung inverser Modelle für die Regelung [Thümmel *et al.* (2005)]. Während ein klassisches Vorwärtsmodell berechnet, welche Wirkung sich aufgrund einer vorgegebenen Ursache ergibt, versucht ein inverses Modell das Gegenteil. Es wird analysiert, welche Ursache zu einer vorgegebenen, gewünschten Wirkung führt. Dieses berechnete Ergebnis wird anschließend als Regeleingang für die Regelstrecke verwendet. Wird die Modellbildung der Regelstrecke akausal, also ohne Vorgabe der Wirkrichtung, durchgeführt, können sowohl das Vorwärtsmodell als auch das inverse Modell unmittelbar aus der selben Modellbeschreibung abgeleitet werden, indem für das inverse Modell die Eingänge als Ausgänge verwendet werden und umgekehrt die Ausgänge als Eingänge. Dies setzt jedoch voraus, dass das inverse Modell ebenfalls stabil ist und sämtliche Komponenten invertierbar sind, was bei der Verwendung von Kennlinien und Tabellen häufig nicht gewährleistet ist [Thümmel *et al.* (2005)].

Weiterhin ist es möglich, das Regelungsproblem als Optimierungsaufgabe aufzufassen. Gesucht ist in diesem Fall eine optimale Steuertrajektorie, die unter Berück-

sichtigung des Systemverhaltens, welches mathematisch in Form von Differentialgleichungen beschrieben wird, und gegebenen Beschränkungen bezüglich der Zustände und Eingangsgrößen eine zuvor definierte Zielfunktion minimiert, die das Regelziel beinhaltet. Die Lösung derartiger dynamischer Optimierungsprobleme ist Bestandteil der Theorie der *Optimalsteuerung* [Plail (1998)]. Wird die Regelung und somit die Optimierung online auf einer Steuerungshardware durchgeführt, wird von *Model Predictive Control* gesprochen. Voraussetzung für die Art der Regelung ist jedoch die Existenz eines Modells der Regelstrecke, das die dynamischen Gleichungen in ausreichender Güte beinhaltet. Durch Generierung von Code aus dem Simulationsmodell kann dieses auf dem Steuergerät für die Optimierung verfügbar gemacht werden.

Durch Verwendung modellbasierter Entwicklungsmethoden ist es möglich, die zuvor erwähnten Probleme bei der Inbetriebnahme, nämlich die Notwendigkeit von Re-Implementierungen bereits vorhandenen Wissens aus vorgelagerten Entwicklungsphasen sowie die Beschränkung, erst nach einer vollständigen Errichtung der Anlage mit der Inbetriebnahme beginnen zu können, zu umgehen. Dies führt gleichzeitig zu einer Reduzierung der Inbetriebnahmezeit. Weiterhin wirkt sich das durchgängige Engineering nicht nur positiv auf die Phase der Inbetriebnahme aus, sondern offenbart Potentiale ebenso für die Phase des Betriebs der Anlage, beispielsweise bei der Verwendung von Modellen zur Diagnose.

Effizienzdefizite bei der Auslegung Jedoch existieren nicht nur bei der Inbetriebnahme Potentiale für eine Reduzierung der Entwicklungszeit. Auch bereits in der Auslegungsphase kann eine Effizienzsteigerung des Entwicklungsprozesses und somit eine Reduzierung der Time-to-Market erreicht werden. Viele Anlagen werden nach dem Prinzip des *trial and error* ausgelegt. Ist das Sollverhalten der Anlage, welches aus den im Rahmen der Anforderungsanalyse erstellten *Lasten-* und *Pflichtenheften* resultiert, ermittelt, wird mit der Auswahl geeigneter Komponenten begonnen. Dies geschieht zumeist anhand von persönlichen Erfahrungswerten. Anschließend werden virtuelle Abbilder dieser Komponenten in einer Simulationsumgebung zu dem Gesamtsystem verknüpft, um die Eignung bezüglich der Sollvorgabe simulativ ermitteln zu können. Für eine Bewertung der dynamischen Eigenschaften ist die Implementierung eines Regelungsalgorithmus innerhalb der Simulationsumgebung notwendig. Sowohl die Auswahl der Komponenten als auch die Parametrierung des Reglers erfolgen manuell durch den Ingenieur. Es wird sukzessive versucht, durch Ändern der Parameter zu einer „optimalen“ Konfiguration zu gelangen.

Bei näherer Betrachtung weist dieses Vorgehen einige Nachteile auf. Vor allem für komplexe Systeme existiert eine hohe Anzahl an veränderlichen Parametern und möglichen Komponenten. Das resultierende Ergebnis ist stark von der Expertise des Applikationsingenieurs abhängig. Weiterhin kann das Auffinden einer zufriedenstellenden Lösung bei einem komplexen System sehr viel Zeit in Anspruch nehmen. In diesem Fall ist die Auswirkung der Variation eines einzelnen Para-

meters auf das Systemverhalten häufig nicht vorhersehbar. Schließlich bleibt auch nach erfolgter Auslegung stets die Frage offen, ob die gefundene Kombination aus Systemkomponenten und Reglerparametern tatsächlich optimal ist, oder ob nicht noch eine bessere Lösung existiert. Soll eine vom Anwender unabhängige optimale Systemauslegung erreicht werden, muss diese automatisiert durchgeführt werden [Kreisselmeier & Steinhauser (1979)].

Die Aufgabe, eine zur Erzielung eines vorgegebenen Sollverhaltens der Anlage optimale Kombination aus Komponenten und Reglerparametern zu ermitteln, stellt mathematisch ein Optimierungsproblem dar, bei dem die Sollvorgabe der Zielfunktion bzw. Kostenfunktion entspricht und die zu ermittelnden Reglerparameter und Systemkomponenten Designvariablen sind. Da Teile der Designvariablen einen reellwertigen Wertebereich haben (Reglerparameter), während einige Designparameter ganzzahlig sind (Systemkomponenten), liegt ein gemischt-ganzzahliges Optimierungsproblem vor. Die Lösung des Optimierungsproblems stellt eine optimale Lösung des Auslegungsproblems dar. Ein wesentlicher Vorteil ist, dass die Auslegungsgüte in diesem Fall unabhängig von der Expertise des Applikationsingenieurs ist. Für sehr komplexe Anlagen mit vielen variablen Größen kann die automatisierte Auslegung weiterhin eine deutliche Zeiteinsparung mit sich bringen.

Auch für eine *Parameteridentifikation* können Optimierungsverfahren sinnvoll eingesetzt werden. Soll beispielsweise für eine bestehende Anlage ein neues Regelungskonzept erprobt werden, kann dies simulativ durchgeführt werden, um Experimente mit der realen Anlage zu vermeiden. Für die Erzielung aussagekräftiger Ergebnisse ist es notwendig, das Verhalten des Simulationsmodells bestmöglich dem Verhalten der realen Anlage anzunähern. Dies geschieht mit Hilfe einer Parameteridentifikation (Kalibrierung des Modells). Während einige Parameter, wie beispielsweise der Durchmesser eines Zylinders oder der Kolbenhub, sehr leicht zu identifizieren sind, ist die Parametrierung von Reibungskoeffizienten innerhalb von Komponenten deutlich komplizierter. Hierüber liegen oftmals keine genauen Informationen vor, da diese unter anderem auch durch Fertigungstoleranzen beeinflusst werden. Rein mathematisch betrachtet führt die Problemstellung, die Parameter eines Modells derart zu bestimmen, dass die Abweichung zwischen simuliertem Verhalten des Modells und einer Messung an der realen Anlage minimiert wird, auf das gleiche Optimierungsproblem wie zuvor beschrieben. Es kann daher mit den gleichen Methoden gelöst werden.

1.2 Stand der Technik

Die Möglichkeit der Verwendung von Simulationsmodellen in der Produktentwicklung ist keine Neuheit und wird in vielen Branchen, wie beispielsweise in der Raumfahrt oder der Automobilindustrie, bereits seit vielen Jahren intensiv genutzt. Vor allem in der Automobilbranche gehören modellbasierte Entwicklungsmethoden seit einigen Jahren zum Standard. In diesem Kontext sind insbesondere Methoden der

automatischen Seriencode-Generierung und der Validierung des Codes an virtuellen Regelstrecken mit Hilfe von MiL-, SiL- bzw. HiL-Simulationen[1] von Bedeutung [Beine (2009)]. Da der auf Automobilsteuergeräten ausgeführte Code zumeist sicherheitskritisch ist, z.B. bei Systemen wie dem elektronischen Stabilitätsprogramm (ESP) oder der automatischen Abstandsregelung (ACC), existieren für die Entwicklung derartiger Anwendungen bereits heute spezielle Normen und Standards.

Von besonderer Bedeutung für die modellbasierte Entwicklung ist die im Jahr 2011 in Kraft getretene, weltweit gültige ISO-Norm 26262 (*Road vehicles – Functional safety*), die die funktionale Sicherheit eines elektronischen Systems in einem Kraftfahrzeug gewährleisten soll. Die Norm ist eine Anpassung der zuvor gültigen IEC-Norm 61508 (*Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme*) und wurde speziell auf die Anforderungen der Automobilbranche angepasst. Die Norm definiert unterschiedliche Sicherheitsanforderungsstufen (*Automotive Safety Integrity Level*, kurz: ASIL), die jeweils prozessunterstützende Maßnahmen und Methoden vorschreiben [Beine (2009)]. Für die Erreichung einer hohen Softwarequalität existieren beispielsweise Codierungsrichtlinien, von denen der von der englischen MISRA[2] entwickelte MISRA-C-Programmierstandard in der ISO 26262 empfohlen wird. Für die Definition und Entwicklung einer offenen und standardisierten Softwarearchitektur bei Steuergerätecode wurde im Jahr 2003 die AUTOSAR[3] Entwicklungspartnerschaft gegründet. Dies ist notwendig, um unter anderem eine einfache Integration von Softwaremodulen von verschiedenen Zulieferern zu ermöglichen.

Für die konkrete Umsetzung eines modellbasierten Engineerings existieren unterschiedliche Softwarewerkzeuge. Zwei weit verbreitete Vertreter sind die von der Firma dSPACE entwickelte MATLAB/Simulink-Erweiterung *TargetLink* sowie das von der ETAS Group entwickelte *ASCET*[4]. Die Erweiterung TargetLink erlaubt die automatische, ISO 26262 konforme Seriencode-Generierung auf Basis einer Untermenge von Simulink- sowie Stateflow-Bausteinen. Es ist mit Hilfe dieses Tools möglich, sowohl ANSI-C-Code als auch prozessoroptimierten Spezialcode zu erzeugen. Insgesamt unterstützt TargetLink den aktuellen AUTOSAR Standard und deckt einen Großteil des MISRA-C-Standards ab [Thomsen (2003)]. Neben TargetLink unterstützt auch ASCET eine automatische prozessoroptimierte Seriencode-Generierung, die ebenfalls konform zu den wichtigsten Standards der Automobilindustrie, wie z.B. AUTOSAR und MISRA-C, ist.

[1] Bei der Model-In-The-Loop-Simulation (MiL-Simulation) wird jeweils ein Simulationsmodell von Regler und Regelstrecke miteinander verbunden, während bei der Software-In-The-Loop-Simulation (SiL-Simulation) aus dem Simulationsmodell des Reglers bereits hardwarenaher Code erzeugt wird, der allerdings noch zusammen mit dem Simulationsmodell der Regelstrecke auf dem Entwicklungsrechner ausgeführt wird. Bei der Hardware-In-The-Loop-Simulation (HiL-Simulation) wird schließlich der Reglercode auf der Zielhardware ausgeführt, während das Steuergerät mit dem virtuellen Streckenmodell interagiert.

[2] **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation

[3] **AUT**omotive **O**pen **S**ystem **AR**chitecture

[4] `http://www.etas.com/ascet`

Modellbasierte Entwicklungsmethoden in Maschinenbau und Automatisierungsindustrie Während der Einsatz modellbasierter Entwicklungsmethoden in einigen Branchen bereits Stand der Technik ist und eine Vielzahl von Normen und Standards existieren, ist der Einsatz derartiger Techniken in anderen Bereichen, wie beispielsweise dem Maschinenbau sowie der Automatisierungsindustrie, weit weniger häufig. Die Entwicklung zukünftiger, hochkomplexer, vernetzter Systeme lässt sich mit den bisher eingesetzten Entwicklungsmethoden jedoch nicht länger bewältigen. Dies ergeben mehrere Studien, die im Kontext der vierten industriellen Revolution (Industrie 4.0) durchgeführt wurden [Gausemeier *et al.* (2013)][Kagermann *et al.* (2013)][Geissbauer *et al.* (2014)]. Die Gründe für die Meidung von modellbasierten Entwicklungsmethoden vor allem in den kleinen und mittleren Unternehmen sind vielfältig. Häufig genannte Punkte sind die fehlende Expertise in den Unternehmen sowie der nicht quantifizierbare Nutzen aufgrund lediglich vereinzelter verfügbarer Werkzeuge zur Unterstützung der modellbasierten Entwicklung [Gausemeier *et al.* (2013)]. Die zur Verfügung stehenden Werkzeuge sind in der Regel kommerzielle Tools, die sich zumeist lediglich auf einen speziellen Abschnitt innerhalb des Entwicklungsprozesses beziehen.

Ein Vergleich der Automatisierungsindustrie und der Automobilindustrie zeigt, dass sich aufgrund der Unterschiede zwischen den Branchen die dort vorhandenen Methoden nicht unmittelbar übertragen lassen. Ein wesentlicher Unterschied ist die Frage, an welcher Stelle die Entwicklung der Steuerungsapplikation erfolgt. In der Automobilindustrie wird die Entwicklung der Steuerungsapplikation in der Regel von großen Unternehmen und großen Entwicklerteams durchgeführt [Reke (2012)]. Anschließend wird die Software gemeinsam mit der Steuerungshardware verkauft, eine Änderung seitens der Kunden wird nicht mehr durchgeführt. Dies trifft auf die Automatisierungsbranche jedoch nicht zu. Die Steuerungshardware wird in der Regel unabhängig von der Applikation verkauft, welche anschließend vom Abnehmer selbst implementiert wird. Wird nun berücksichtigt, dass sämtliche in der Automobilindustrie verwendeten Werkzeuge kommerzielle Tools sind und daher hohe Lizenzgebühren für die Nutzung entrichtet werden müssen, wird ersichtlich, dass diese Lizenzkosten für die großen Unternehmen der Automobilindustrie von nachrangiger Bedeutung sind, während diese für viele Firmen der Automatisierungsbranche ein Problem darstellen. Vor allem für kleine und mittlere Unternehmen (KMU) ist der Kauf von Lizenzen für kommerzielle Plattformen häufig nicht rentabel, sodass modellbasierte Entwicklungsmethoden nicht zugänglich sind.

Eine häufig eingesetzte Lösung zur Integration von Simulationen in die Entwicklung basiert, wie bereits zuvor beschrieben, auf der kommerziellen Softwareumgebung MATLAB/Simulink der Firma Mathworks[1]. Während der Einsatz dieses Frameworks in der Automobilindustrie weit verbreitet ist, ist die Verwendung im Maschinenbau bzw. der Automatisierungsindustrie häufig problematisch und weist einige

[1] `http://de.mathworks.com/index.html`

Nachteile auf. Der erste Nachteil sind die bereits zuvor angesprochenen hohen Lizenzkosten, sodass vor allem KMUs nicht in der Lage sind, die Software einzusetzen. Des Weiteren handelt es sich um eine abgeschlossene Softwareumgebung, wodurch der Austausch von Modellen grundsätzlich erschwert wird, da die erstellten Modelle zunächst nur innerhalb der Umgebung verwendet werden können. Für die Automobilindustrie und deren Standards, beispielsweise den AUTOSAR-Standard, der Schnittstellen für einen Austausch von Modellen definiert, stehen Erweiterungen zur Verfügung, ein von der Automobilindustrie unabhängiger, allgemeiner Modellaustausch wie beispielsweise das *Functional Mockup Interface* (FMI, [Blochwitz *et al.* (2011)]) wird jedoch standardmäßig nicht unterstützt. Weiterhin ist die Verwendung von MATLAB/Simulink aufgrund des signalflussbasierten Modellierungskonzeptes für Regler gut geeignet, für die Modellbildung physikalischer Systeme eignet sich der Ansatz nur bedingt [Schmitz (2007)]. In diesem Fall müssen unterschiedliche Simulationsumgebungen für den Regler und die Regelstrecke verwendet werden.

Für eine ganzheitliche, durchgängige Entwicklung von Systemen steht die *3DEXPERIENCE Plattform* (3DXP) der Firma Dassault Systèmes[1] zur Verfügung. Diese ermöglicht es, bereits Anforderungen an das System mit in das Projekt aufzunehmen. Mit Hilfe dieser Anforderungen kann das mechanische CAD-Modell innerhalb des *CATIA*- bzw. des *SolidWorks*-Moduls der 3DXP aufgebaut werden. Parallel dazu können elektrische Komponenten wie Kabel in das CAD-Modell integriert werden, um anschließend einen elektrischen Schaltplan abzuleiten. Aus dem mechanischen CAD-Modell lässt sich zur Analyse der Dynamik automatisch ein Modelica-Modell erzeugen. Mit Modelica wird innerhalb der 3DXP eine offene, standardisierte Modellierungssprache verwendet, die sehr gut für die Modellierung physikalischer Systeme geeignet ist. Weiterhin sind über die Kompatibilität zum FMI-Standard wesentliche Voraussetzungen für eine Wiederverwendung der Modelle sowie einen Modellaustausch über Toolgrenzen hinweg erfüllt. Die 3DXP hat jedoch auch vor allem für kleinere und mittlere Unternehmen den Nachteil hoher Lizenzkosten. Weiterhin unterstützt die Softwareumgebung lediglich die Entwicklung der Anlage bis zum Abschluss der Auslegungsphase. Ein Transfer des vorhandenen Wissens über die Auslegungsphase hinweg in die Inbetriebnahme sowie den Betrieb ist mit diesem Framework nicht möglich. Gerade an dieser Stelle lässt sich jedoch eine deutliche Effizienzsteigerung erzielen.

Insgesamt steht momentan keine Lösung bereit, die ein durchgängiges Engineering von der Auslegung bis in den Betrieb ermöglicht und somit in der Lage ist, die Entwicklung komplexer Produkte zu unterstützen. Die Entwicklung heutzutage basiert auf einer Reihe unterschiedlicher Tools, die jeweils für spezielle Einsatzgebiete entwickelt wurden. Der Austausch von Daten zwischen den Tools, zum Beispiel die Überführung von Modellwissen aus einem 3D-CAD-Modell in ein dynamisches Modell, ist aufgrund fehlender Standards und einer Vielzahl unterschiedlicher kommerzieller Tools kaum möglich. Weiterhin sind die vorhandenen Lösungen in der

[1] `http://www.3ds.com/de/`

Regel kommerzielle Tools, sodass sich eine starke Abhängigkeit gegenüber den vertreibenden Firmen ergibt, da die Modelle lediglich innerhalb eines speziellen Tools verwendet werden können und ohne dieses Tool nutzlos sind. So ist es bei der zuvor beschriebenen 3DXP zwar möglich, aus einer 3D-CAD-Beschreibung ein dynamisches Modelica-Modell zu erstellen, jedoch ist diese Funktionalität ausschließlich bei Verwendung des ebenfalls von der Firma Dassault Systèmes vertriebenen CAD-Tools CATIA verfügbar.

Soll eine automatische Auslegung des Systems mithilfe von mathematischen Optimierungsmethoden durchgeführt werden, so kann diese nach aktuellem Stand der Technik ebenfalls nicht unmittelbar in einen durchgängigen Workflow eingebettet werden. Stattdessen steht auch hierzu eine Vielzahl von zumeist kommerziellen Tools bereit, die mit vor- bzw. nachgelagerten Schritten der Entwicklung nicht kompatibel sind. Eine Möglichkeit ist die Verwendung des Optimierungstools *ModelOPT* der Firma XRG[1]. Dieses Tool ermöglicht die Optimierung von Systemen auf Basis von Modelica-Modellen. Jedoch wird für die Simulation zwingend das kommerzielle Modelica-Tool *Dymola* benötigt, die Verwendung frei erhältlicher Modelica-Umgebungen ist nicht möglich. Weiterhin können lediglich kontinuierliche Optimierungsprobleme gelöst werden, gemischt-ganzzahlige Optimierungsprobleme, wie sie in der Regel bei der Suche nach einer optimalen Kombination von Systemkomponenten und Reglerparametern auftreten, werden nicht unterstützt. Die Suche nach optimalen Komponenten stellt jedoch einen wesentlichen Fall bei der automatischen Auslegung von Systemen dar. Innerhalb des *OpenModelica*-Projektes, welches die Entwicklung eines freien Modelica-Compilers vorantreibt, existiert das Tool *OMOptim*[2] zur Optimierung [Thieriot *et al.* (2011)]. Jedoch unterstützt auch dieses Tool nicht die Optimierung beliebiger gemischt-ganzzahliger Optimierungsprobleme. Des Weiteren wird OMOptim bereits seit längerer Zeit nicht weiterentwickelt.

Damit auch kleine und mittlere Unternehmen zukünftig in der Lage sind, die steigenden Anforderungen an die Produktentwicklung zu bewältigen, reichen die heutigen Softwarewerkzeuge zur Unterstützung nicht aus. Eine vom Heinz Nixdorf Institut der Universität Paderborn in Zusammenarbeit mit dem Fraunhofer Institut für Produktionstechnologie IPT sowie der UNITY AG durchgeführte Studie bestätigt dies [Gausemeier *et al.* (2013)]. Abbildung 1.5 stellt dar, in welchen Bereichen zurzeit die größten Defizite liegen. Das Fehlen durchgängiger Werkzeugketten sowie eine zu niedrige Methodenkompetenz sind die häufigsten Punkte, die von Firmen genannt wurden. Zusätzlich muss auch die Akzeptanz für neue Herangehensweisen in den Unternehmen gesteigert werden, die häufig bisher neuen Methoden skeptisch gegenüberstehen. Dies kann dadurch erreicht werden, dass sich die Methoden, Vorgehensweisen und Werkzeuge an den Nutzern orientieren, sodass der Benutzer sie nicht als Last, sondern als Unterstützung empfindet [Gausemeier *et al.* (2013)]. Der Arbeitskreis Industrie 4.0 kommt in seinem Abschlussbericht weiterhin zu dem

[1] `http://xrg-simulation.de/de/produkte/applications/modelopt`
[2] `https://openmodelica.org/research/omoptim`

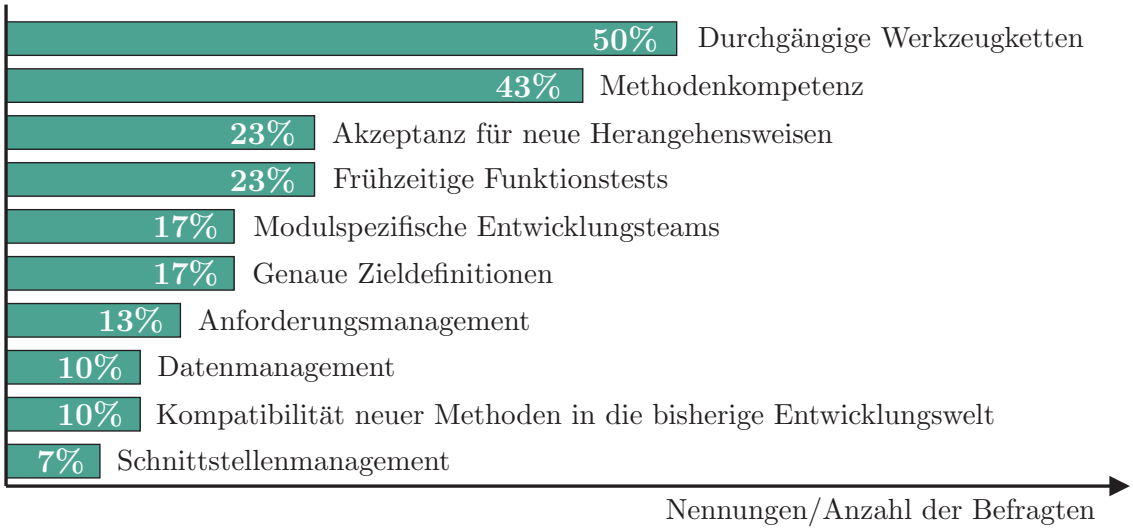

Abbildung 1.5: Was muss getan werden? [Gausemeier *et al.* (2013)]

Schluss, dass eine firmenübergreifende Vernetzung und Durchgängigkeit der Entwicklung nur auf Basis offener Standards gelingen kann [Kagermann *et al.* (2013)].

1.3 Zielsetzung und Aufbau der Arbeit

Die vorliegende Arbeit begegnet den zuvor genannten Problemen und beschäftigt sich mit Methoden zur Effizienzsteigerung innerhalb des Produktentwicklungsprozesses und deren Umsetzung für einen Einsatz in der Praxis. Die Methoden betreffen die Phasen der Auslegung und der Inbetriebnahme bei der Entwicklung mechatronischer Systeme im Maschinenbau und der Automatisierungsindustrie. Die Effizienzsteigerung sorgt für eine Verkürzung der Time-to-Market und somit eine Stärkung der eigenen Marktposition und resultiert hauptsächlich aus der konsequenten Umsetzung eines durchgängigen, modellbasierten Engineerings.

Ein umfassendes, durchgängiges Engineering lässt sich allerdings, wie zuvor ausführlich beschrieben, nicht auf Basis kommerzieller Tools umsetzen, da diese zumeist abgeschlossen sind und ein Modellaustausch - wenn überhaupt - nur eingeschränkt möglich ist. Aus diesem Grund ist die Umsetzung der Methoden mit Hilfe von offenen Standards eine wesentliche Anforderung, um einen unternehmensübergreifenden Einsatz und Austausch sowie eine Wiederverwendung von Modellen zu ermöglichen. Gleichzeitig können dadurch auch kleine und mittlere Unternehmen von den modellbasierten Entwicklungsmethoden profitieren. Für einen unkomplizierten Einsatz in der Industrie ist es erforderlich, ausschließlich bestehende Industriehardware zu verwenden, ohne auf spezielle zusätzliche Hardware zurückgreifen zu müssen, die den Einsatz nur in speziellen Fällen ermöglicht. Der Fokus innerhalb dieser Arbeit liegt, wie bereits erwähnt, auf den Phasen der simulativen Auslegung von Systemen

und der Inbetriebnahme. Die modellbasierte Entwicklung ermöglicht jedoch auch deutliche Verbesserungen innerhalb der Phase des Betriebs, beispielsweise in Form von *Smart Services*. Durch neuartige, hochvernetzte Maschinen stehen heutzutage Unmengen an Daten bereit. Eine Aufbereitung und intelligente Nutzung dieser Daten - unter anderem auch in Kombination mit den Systemmodellen - ermöglicht es Anbietern, Ressourcen zu schonen und die eigene Wettbewerbsfähigkeit zu steigern [Bauernhansl *et al.* (2014)]. Diese Techniken sind nicht Bestandteil dieser Arbeit.

Im Rahmen dieser Arbeit wird ein durchgängiger, modellbasierter Entwicklungsprozess auf Basis von offenen Standards vorgestellt. Für eine Umsetzung in der Praxis werden folgende Komponenten bereitgestellt:

- Entwicklung eines Optimierungsmoduls für eine automatisierte Auslegung von Systemen auf Basis von Modelica-Modellen, wobei ein Schwerpunkt auf der Unterstützung von gemischt-ganzzahligen Optimierungsproblemen liegt, wie sie bei der automatischen Auswahl von optimalen Systemkomponenten auftreten

- Entwicklung einer Werkzeugkette zur automatischen Generierung von ausführbarem Code aus Modelica-Modellen für die Verwendung auf Industriesteuerungen

- Erweiterung des freien Simulationskern des OpenModelica-Projektes um numerische Methoden für die Durchführung von Echtzeitsimulationen

- Entwicklung und Bereitstellung von offenen Schnittstellen zur Einbindung von Industriesteuerungen in Simulationsumgebungen zur Ermöglichung einer virtuellen Inbetriebnahme

Die Arbeit gliedert sich dabei wie nachfolgend beschrieben. In Kapitel 2 werden zunächst Grundlagen behandelt, die für die weiteren Betrachtungen notwendig sind. Dazu wird in Abschnitt 2.1 die objektorientierte Modellbildung behandelt. Es werden zunächst die Grundsätze dieser Methode vorgestellt, welche die Vorteile der Verwendung dieser Art der Modellbildung untermauern. In Kapitel 2.2 werden anschließend numerische Integrationsverfahren betrachtet, die zur Lösung der im Kontext der objektorientierten Modellbildung auftretenden Systeme gewöhnlicher Differentialgleichungen benötigt werden. Abschließend beschäftigt sich Abschnitt 2.3 mit mathematischen Verfahren zur Lösung von Optimierungsproblemen. Hierbei wird insbesondere auf Verfahren zum Lösen von rein reellwertigen sowie gemischtganzzahligen Optimierungsproblemen eingegangen.

Die Vorstellung des modellbasierten Entwicklungsprozesses und die Umsetzung der Methoden erfolgt in Kapitel 3. Dazu wird in Abschnitt 3.1 zunächst die VDI Richtlinie 2206, die heutzutage die Grundlage der Entwicklung mechatronischer Systeme darstellt, betrachtet. Es werden schließlich diejenigen Punkte identifiziert, die im

Sinne einer Verbesserung des Entwicklungsprozesses in der bisherigen Norm entweder fehlen oder erweitert werden können. Aus dieser Analyse lassen sich unmittelbar der im Rahmen dieser Arbeit vorgestellte modellbasierte Entwicklungsprozess und die zur Umsetzung benötigten Bausteine ableiten. Diese werden anschließend in der Reihenfolge entsprechend der Verwendung im Entwicklungsprozess vorgestellt. Abschnitt 3.2 stellt Möglichkeiten vor, wie unter Verwendung von mathematischen Optimierungsverfahren Zeit- und somit Kosteneinsparungen bei der Auslegung erzielt werden können. Im folgenden Abschnitt 3.3 werden die Modelle aus der Auslegungsphase in die Inbetriebnahme überführt und dort wiederverwendet. Es wird eine auf offenen Standards basierende Werkzeugkette vorgestellt, die es ermöglicht, aus beliebigen Modellen, die auf einer toolunabhängigen, standardisierten Modellierungssprache basieren, lauffähigen Code zu erzeugen und diesen auf Industriesteuerungen auszuführen. Die Ausführung des Codes auf der Industriehardware erfolgt in Echtzeit, was besondere echtzeitfähige numerische Integrationsverfahren zur Lösung der Modellgleichungen erfordert. Diese Thematik wird ebenfalls in diesem Abschnitt behandelt. Bevor die Steuerungsapplikation an der realen Anlage getestet wird, ist es sinnvoll, diese in einer sicheren Umgebung virtuell zu testen. Eine derartige Einbindung von Hardware in Softwareumgebungen wird als Hardware-In-The-Loop-Simulation (HiL) bezeichnet. Im Abschnitt 3.4 wird eine Möglichkeit vorgestellt, wie HiL-Simulationen zur Funktionsüberprüfung des Reglercodes durchgeführt werden können.

Der Entwicklungsprozess und die vorgestellten Methoden werden in Kapitel 4 auf zwei Beispiele angewendet, um die Funktionsweise zu validieren. An dem Beispiel eines Gleichlaufs zweier hydraulischer Achsen zur Entwicklung einer hydraulischen Presse wird in Abschnitt 4.1 von der Auslegung bis zum Betrieb gezeigt, wie der Produktentwicklungsprozess in der Praxis angewendet werden kann. In Abschnitt 4.2 wird veranschaulicht, wie unter Zuhilfenahme der Methoden der virtuellen Inbetriebnahme und der Codegenerierung eine komplexe Flaschenabfüllanlage ausgelegt, getestet und virtuell in Betrieb genommen wird. Dazu werden die entwickelten Bausteine der Codegenerierung und virtuellen Inbetriebnahme mit der komplexen, kommerziellen 3DEXPERIENCE Plattform kombiniert, wodurch der besondere Vorteil der im Rahmen dieser Arbeit entwickelten Lösung auf Basis offener Standards deutlich wird.

Das abschließende Kapitel 5 fasst die Arbeit zusammen und gibt einen Ausblick auf weitere Arbeiten. Insbesondere wird geprüft, inwiefern die Zielsetzung der Arbeit erreicht wurde und aufgezeigt, an welchen Stellen für weitere Untersuchungen angesetzt werden sollte.

KAPITEL 2

Grundlagen

Ein effizienter Einsatz modellbasierter Entwicklungsmethoden lässt sich am einfachsten durch die Verwendung objektorientierter Modellierungsmethoden erreichen. Daher wird in diesem Grundlagenkapitel zunächst die objektorientierte Modellierung von der signalflussbasierten Modellierung abgegrenzt. Anschließend wird der bekannteste Vertreter dieser Klasse, die Modellierungssprache Modelica, eingeführt und der typische Ablauf einer Simulation beschrieben. Der zweite Teil des Kapitels beschäftigt sich mit numerischen Methoden zur Lösung von gewöhnlichen Differentialgleichungen, wie sie bei der Modellierung von technischen Systemen auftreten. Abschließend wird auf Grundlagen mathematischer Optimierungsverfahren eingegangen. Neben Methoden für reellwertige Optimierungsprobleme stehen vor allem Methoden zur Lösung gemischtganzzahliger Optimierungsprobleme im Fokus.

2.1 Objektorientierte Modellbildung

Die Entwicklung komplexer mechatronischer Systeme macht eine domänenübergreifende Denkweise notwendig. Während früher die einzelnen Subsysteme jeweils in einer speziellen domänenspezifischen Simulationsumgebung entwickelt und getestet wurden, ist es für den mechatronischen Entwicklungsprozess notwendig, die Modelle der einzelnen Subsysteme in einer gemeinsamen Simulationsumgebung verfügbar zu haben. Damit lässt sich nicht nur die Anzahl der Iterationen innerhalb des Produktentwicklungsprozesses reduzieren, sondern gleichzeitig durch die schnelle und unkomplizierte Durchführung umfassender, domänenübergreifender Funktionstests

auch die Produktqualität verbessern [Drogies (2005)]. Die domänenspezifischen Subsysteme werden bisher zumeist von Experten der jeweiligen Domäne *signalflussbasiert* modelliert. Hierbei wird das Verhalten des betrachteten Systems durch mathematische Gleichungen beschrieben. Diese Gleichungen werden anschließend mit Hilfe von Funktionsblöcken nachgebildet. Ein typisches Beispiel für diese Art der Modellbildung ist die Modellierung in MATLAB/Simulink.

Da anhand der modellierten mathematischen Gleichungen nur schwer auf die physikalische Struktur des Systems geschlossen werden kann, ist eine Anpassung eines Submodells bei einer Erweiterung des Systems aufwendig und kann zumeist nur von Domänenexperten durchgeführt werden. Innerhalb einer gemeinsamen Simulationsplattform ist es daher notwendig, dass die Modelle so beschaffen sind, dass sie auch auch von Experten anderer Domänen verwendet werden können, ohne dass detailliertes Wissen über die zugrundeliegenden Gleichungen notwendig ist. Ein weiterer wichtiger Aspekt ist die Wiederverwendbarkeit der Modelle. Häufig beinhalten unterschiedliche Systeme immer wieder die gleichen Basiskomponenten. Zur Reduzierung der Entwicklungszeit und somit zur Steigerung der Effizienz des Entwicklungsprozesses ist es wünschenswert, häufig wiederkehrende Basiskomponenten in geeigneter Form innerhalb einer Modellbibliothek verfügbar zu machen, sodass diese Komponenten nicht für jedes System neu modelliert werden müssen.

Ein Ansatz, der die eben genannten Punkte umsetzt, ist die *objektorientierte* Modellierung. Die Grundlage bildet eine einheitliche, domänenübergreifende Modellierungssprache. Insgesamt kann dadurch eine höhere Wartbarkeit, eine höhere Flexibilität sowie aufgrund der Möglichkeit der Verwendung einer graphischen Modellierungsoberfläche eine hohe topologische Ähnlichkeit der Modelle mit der physikalischen Struktur des Systems erreicht werden. Der Unterschied zwischen signalflussbasierter und objektorientierter Modellierung im Hinblick auf die Erweiterbarkeit von Modellen wird bereits für kleine Systeme deutlich. Dies wird in [Kampfmann (2014)] am Beispiel eines Ein- bzw. Zweimassenschwingers gezeigt.

Im folgenden Kapitel 2.1.1 werden die kennzeichnenden Eigenschaften objektorientierter Modellierungssprachen vorgestellt. Der bedeutendste Vertreter dieser Klasse, die Sprache *Modelica*, wird in Kapitel 2.1.2 behandelt.

2.1.1 Grundsätze objektorientierter Modellbildung

Auch wenn die grundlegenden Konzepte bereits älter sind, wurde der Begriff der *objektorientierten Modellbildung* in [Nilsson (1989)] erstmals im Jahr 1989 verwendet. Die erste objektorientierte Programmiersprache, die zur Simulation physikalischer Systeme entwickelt und eingesetzt wurde, ist *Simula 67*. Sie gilt außerdem als erste objektorientierte Programmiersprache überhaupt und wurde von Ole-Johan Dahl und Kristen Nygaard entwickelt [Dahl *et al.* (1968)]. Im Laufe der Zeit wurden verschiedene objektorientierte Modellierungssprachen wie *Allan*, *Dymola*, *NMF*, *Ob-*

jectMath, *Omola*, *SIDOPS+* und *Smile* entwickelt. Aus diesen Sprachen ist anschließend der heute wichtigste Vertreter *Modelica* entstanden, um die einzelnen Sprachen miteinander zu vereinen und einen Standard zu schaffen [Jeandel & Boudaud (1997)]. Weitere Ansätze objektorientierter Modellierungssprachen finden sich in [Runge (1977)] und [Elmqvist (1978)].

Obwohl also bereits seit längerer Zeit Beispiele für objektorientierte Modellierungssprachen existieren, gibt es keine eindeutige Definition. Die Grundsätze objektorientierter Modellierungsmethoden können aber aus denen objektorientierter Programmiersprachen abgeleitet werden. Es gibt vielmehr eine Liste von Kriterien, die zur Überprüfung herangezogen werden kann, ob es sich um eine objektorientierte Modellierungssprache handelt [Andersson (1994)]. Die Beispiele zu den einzelnen Kriterien sind dabei an [Mikelsons (2012)] angelehnt.

Verwendung deklarativer Modelle Die in einem Modell verwendeten Gleichheitszeichen bezeichnen keine Zuweisung, sondern eine Äquivalenz. Das bedeutet, dass es unerheblich ist, ob eine Gleichung in der Form

$$F = c \cdot \Delta x \tag{2.1}$$

oder in der Form

$$\Delta x = \frac{F}{c} \tag{2.2}$$

angegeben wird. Während in signalflussbasierten Modellen die Kausalität bereits im Vorfeld vom Anwender festgelegt werden muss, kann im objektorientierten Fall akausal modelliert werden. Die Kausalisierung wird in diesem Fall nicht vom Anwender durchgeführt. Stattdessen ist das Umformen der Gleichungen zur Erlangung der benötigten Kausalität eine der wichtigsten Aufgaben während des Simulationsvorganges.

Modularität Gesamtmodelle bestehen in der Regel nicht aus einer einzigen Hierarchiestufe, sondern sind modular aufgebaut. Das bedeutet, dass sie aus einer Vielzahl von Submodellen aufgebaut sind, die selbst wiederum aus mehreren Submodellen bestehen können. Auf diese Weise wird das Gesamtsystem auf seine Grundkomponenten heruntergebrochen, welche anschließend zu dem Gesamtmodell verknüpft werden. Die Gleichungen sind lediglich auf der untersten Stufe, in den jeweiligen Grundkomponenten, enthalten. Ein komplexes Modell einer mobilen Arbeitsmaschine, z.B. eines Baggers, beinhaltet unter anderem ein Antriebsmodell. Das Antriebsmodell selbst besteht wiederum aus verschiedenen Submodellen wie einem Zylindermodell sowie dem Modell einer Pumpe. Die mathematischen Gleichungen des Zylinders sowie der Pumpe sind jeweils in den Basiskomponenten Zylinder sowie Pumpe implementiert. Das Gesamtverhalten ergibt sich aus dem Zusammenwirken

der einzelnen Grundkomponenten. Auf diese Weise ist es möglich, die Topologie des betrachteten Systems auch im Modell nachzubilden.

Abstraktion Die Eigenschaften der Abstraktion und Modularität sind eng miteinander verknüpft. Das Konzept der Abstraktion sorgt für eine Kapselung des physikalischen Wissens. Jedes Modell besteht aus einer Modellbeschreibung in Form von mathematischen Gleichungen und Schnittstellen, über die von außen auf das Modell zugegriffen werden kann. Auf Basis dieser Schnittstellen sind die Basismodelle miteinander verbunden und können Daten austauschen. Diese Art der Modellierung ermöglicht es, dass auch Laien entsprechende Komponenten verwenden können, weil das Wissen über die Komponente in dieser gekapselt ist und der Anwender das Modell lediglich über seine Schnittstellen anspricht. Um Module unterschiedlicher Domänen miteinander verbinden zu können, müssen diese Schnittstellen physikalisch motiviert sein. Jeglichen physikalischen Prozessen liegen gewisse Erhaltungs- und Bilanzgleichungen zugrunde. Für die Schnittstellen wird daher das Prinzip der Energieerhaltung verwendet. Die Schnittstellen bestehen aus einem Variablenpaar, einer *Across*- und einer *Through*-Variablen, deren Produkt die momentane Leistung darstellt [Wellstead (1979)]. Die Erhaltungsgleichungen lauten allgemein

$$a_1 = a_2 = \cdots = a_n \tag{2.3}$$

für die Across-Variablen sowie

$$\sum_{i=1}^{n} t_i = 0 \tag{2.4}$$

für die Through-Variablen [Schramm (2013)]. Tabelle 2.1 enthält eine Übersicht von Across- und Through-Variablen für einige physikalische Domänen. Da dieses Konzept für jede Domäne gleichermaßen gilt, erlaubt es eine Modellierung von domänenübergreifenden Systemen anhand des Energieflusses.

Domäne	Across-Variable	Through-Variable
Mechanik	Geschwindigkeit	Kraft
Elektrotechnik	Spannung	Strom
Hydraulik	Druck	Volumenstrom
Thermodynamik	Temperatur	Entropiestrom
Magnetismus	magnetomotorische Kraft	Flussänderung

Tabelle 2.1: Across- und Through-Variablen für verschiedene Domänen, frei nach [Schramm (2013)]

Klassen Analog zur objektorientierten Programmierung wird auch bei der objektorientierten Modellierung zwischen Klassen und Instanzen unterschieden. Während eine Klasse eine Abstraktion des Systems ist, beschreibt eine Instanz eine konkrete Realisierung dieses Systems. Von einer Klasse können dabei unterschiedliche Instanzen angelegt werden, die jeweils einen unterschiedlichen Satz von Attributen besitzen. Beispielsweise könnte die Klasse *Kraftfahrzeug* die Attribute *Fahrzeugmasse*, *Schwerpunkt* und *Fahrzeugmaße* besitzen. Eine konkrete Instanz dieser Klasse ist z.B. der VW Golf mit den entsprechenden Attributwerten. Das Konzept der Klassen und Instanzen erhöht die Wiederverwendbarkeit. Ein Basismodell braucht lediglich einmal implementiert zu werden und kann bei Bedarf an mehreren Stellen instanziiert werden.

Vererbung Vererbung bezeichnet die Möglichkeit, eine Unterklasse von einer bestehenden Basisklasse abzuleiten. Beispielsweise kann die Klasse *Elektrofahrzeug* von der bestehenden Basisklasse *Fahrzeug* abgeleitet werden. Diese erbt damit sämtliche Attribute und Funktionen der Basisklasse. Weiterhin besteht die Möglichkeit, weitere, für diese abgeleitete Klasse spezifische Funktionen und Attribute hinzuzufügen, z.B. das lediglich Elektrofahrzeuge betreffende Attribut *Batteriekapazität*.

2.1.2 Modelica

Wie bereits im vorherigen Abschnitt beschrieben, existieren die Konzepte der objektorientierten Modellierung bereits seit den siebziger Jahren. Es entstanden viele unterschiedliche Modellierungssprachen, die sich jedoch zu dieser Zeit noch nicht durchsetzen konnten, was im Wesentlichen zwei Ursachen hatte. Zum einen war die Rechenleistung damaliger Computer in der Regel nicht ausreichend, um komplexe Systeme zu lösen, zum anderen mangelte es an robusten numerischen Verfahren, die für die Lösung der Modellgleichungen erforderlich sind. Dies änderte sich im Laufe der neunziger Jahre, als auch Standardcomputer ausreichend Rechenleistung besaßen, um komplexe Modelle zu simulieren. Gleichzeitig ist in der Zwischenzeit die Entwicklung robuster numerischer Verfahren deutlich vorangeschritten. Aus dem Ziel, die vielen unterschiedlichen Sprachen zu vereinheitlichen, entstand im September 1997 die erste Spezifikation der Sprache *Modelica* [Fritzson & Engelson (1998)]. Drei Jahre später wurde die gemeinnützige Modelica Association gegründet, die seitdem die Weiterentwicklung der Sprache vorantreibt. Ferner wird auch die *Modelica Standard Library (MSL)*, die offizielle Standardbibliothek mit Basiskomponenten unterschiedlicher Domänen, von der Modelica Association verwaltet.

Da es sich bei Modelica um eine Sprache zur Modellierung handelt, wird zur Simulation der Modelle ein Compiler benötigt, der lauffähigen Code aus der Modellbeschreibung erzeugt. Mittlerweile existieren unterschiedliche Modelica-Compiler, wobei der Großteil der Compiler Teil umfangreicher, kommerzieller Simulationsplattformen ist. Diese Simulationsplattformen besitzen zusätzlich eine graphische

Oberfläche zur Modellierung. Die bekanntesten kommerziellen Modelica-basierten Simulationstools sind *Dymola* [Brück *et al.* (2002)], *SimulationX*, *Wolfram SystemModeler* und *MapleSim* [Hřebíček *et al.* (2008)].

Neben den zuvor genannten kommerziellen Compilern existieren weiterhin Open Source Compiler. Die bekanntesten Vertreter entstammen dem *OpenModelica* Projekt [Fritzson *et al.* (2005)] [Fritzson *et al.* (2002)] sowie dem *JModelica.org* Projekt [Åkesson *et al.* (2009)]. Für die Entwicklung und Verwaltung des OpenModelica-Compilers ist das OpenModelica Konsortium zuständig, welches aus unterschiedlichen Firmen aus der Industrie (Bosch Rexroth, Siemens, Wolfram MathCore, ABB) sowie Universitäten (Universität Linköping, FH Bielefeld, Polytechnikum Mailand) besteht. Im Rahmen dieser Arbeit wird der OpenModelica-Compiler verwendet. Der Aufbau und die Funktionsweise des OpenModelica-Compilers wird im nachfolgenden Abschnitt erläutert.

2.1.3 Aufbau des OpenModelica-Compilers

Der Compiler ist der wichtigste Bestandteil einer Simulationsplattform und dient dazu, aus der Modellbeschreibung in Modelica eine lauffähige Simulation zu erzeugen. Die Struktur eines Modelica-Compilers ist anhand des OpenModelica-Compilers in Abbildung 2.1 dargestellt.

2.1.3.1 Struktur des OpenModelica-Compilers

In einem ersten Schritt wird im *Frontend* aus der Modellbeschreibung das flache Modell erstellt. In diesem Schritt werden die Gleichungen sämtlicher Komponenten zu einem großen Gleichungssystem zusammengefasst, wodurch die objektorientierten Strukturen aus dem Modell entfernt werden [Frenkel (2014)]. Aus diesem Schritt resultiert ein System impliziter differential-algebraischer Gleichungen, welches die Form

$$\mathbf{F}(\mathbf{x}, \dot{\mathbf{x}}, t) = 0 \tag{2.5}$$

besitzt. Es könnte versucht werden, das Gleichungssystem in dieser Form mit Hilfe eines DAE[1]-Lösers zu lösen, jedoch ist dieses Vorgehen, sofern das Gleichungssystem auf diese Weise überhaupt gelöst werden kann, äußerst ineffizient [Mikelsons (2012)]. Stattdessen wird versucht, das DAE-System mit Hilfe geeigneter Manipulationen in ein System gewöhnlicher Differentialgleichungen zu überführen, welches deutlich effizienter gelöst werden kann.

Die Manipulation der Gleichungen wird im *Backend* des Compilers durchgeführt. Auf die wichtigsten Operationen während der Gleichungsmanipulation sowie auf mögliche Probleme bei der Transformation wird in Abschnitt 2.1.3.2 eingegangen.

[1] **D**ifferential **A**lgebraic **E**quation

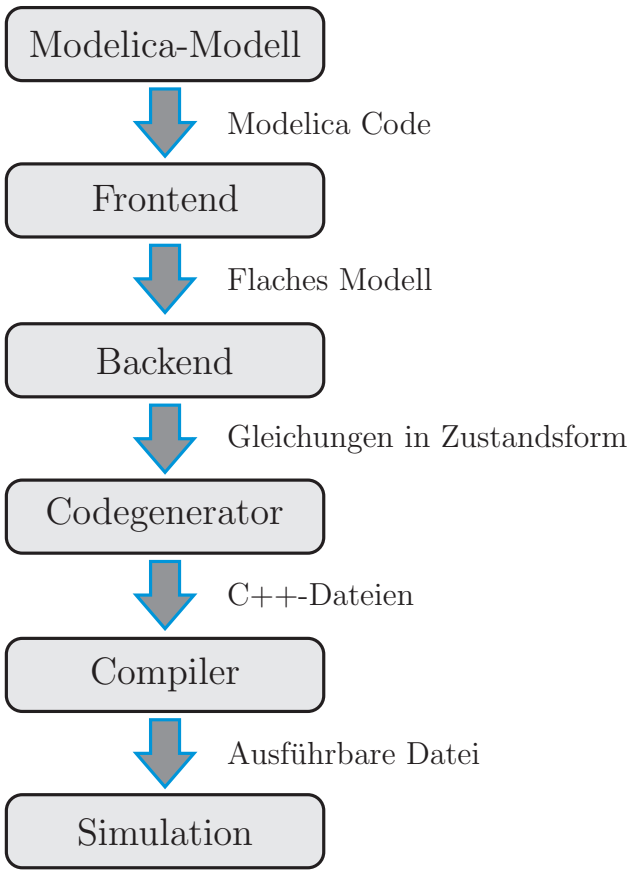

Abbildung 2.1: Struktur des OpenModelica-Compilers, frei nach [Sjölund (2015)]

Aus den effizient lösbaren Gleichungen, die schließlich in der Zustandsform

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \tag{2.6}$$

vorliegen, wird in einem dritten Schritt ausführbarer Code erzeugt. Der *Codegenerator* ist in der Templatesprache Susan geschrieben [Fritzson *et al.* (2009)]. Durch den templatebasierten Aufbau kann der Codegenerator auf einfache Weise auf spezielle Bedürfnisse angepasst werden. Im OpenModelica-Compiler sind standardmäßig mehrere Codegeneratoren verfügbar, beispielsweise besteht die Möglichkeit der Generierung von C-Code sowie von C++-Code. Im Rahmen dieser Arbeit wird die C++-Codegenerierung verwendet. Für eine detaillierte Beschreibung der einzelnen Schritte während der Codegenerierung sei auf [Lindberg (2010)] verwiesen.

Der letzte Schritt besteht darin, den generierten Code mit Hilfe eines *Compilers* für das Betriebssystem zu kompilieren, auf dem der Code ausgeführt werden soll. Das Modell enthält jedoch nur die Gleichungen des Systems. Zum Lösen der Gleichungen werden numerische Lösungsverfahren benötigt. Die Steuerung des Ablaufs der Simulation wird von einem Simulationskern übernommen, welcher sich neben dem Bereit-

stellen der Lösungsverfahren beispielsweise auch um die Datenhaltung kümmert. Im Rahmen dieser Arbeit wird der in C++ geschriebene Simulationskern, der ebenfalls Teil des OpenModelica-Projektes ist, verwendet [Mikelsons & Worschech (2012)]. Das Ergebnis ist ein ausführbares Programm, welches beim Starten die *Simulation* durchführt. Die Simulationsergebnisse werden in einer Ausgabedatei gespeichert.

2.1.3.2 Symbolische Manipulation differential-algebraischer Gleichungen

Das objektorientiert modellierte System wird im Frontend in ein DAE-System transformiert. Die symbolische Manipulation dieser differential-algebraischen Gleichungen im Backend sorgt dafür, dass das DAE-System in ein wesentlich effizienter lösbares System aus gewöhnlichen Differentialgleichungen überführt wird. Da die Manipulation automatisiert geschieht, sind leistungsfähige Algorithmen notwendig, die gleichermaßen auf jedes System angewendet werden können. Das Vorgehen ist stets identisch. In einem ersten Schritt wird ein bipartiter Graph erzeugt, der die Gleichungen des Modells und die darin auftretenden Variablen in Verbindung bringt. Auf diesem bipartiten Graphen wird anschließend versucht, mit Hilfe eines *Matching-Algorithmus* eine Zuordnung zwischen den im Modell vorhandenen Gleichungen und Variablen zu bestimmen. Nach diesem Schritt ist bekannt, welche Gleichung nach welcher Variablen aufgelöst werden muss. In welcher Reihenfolge diese Gleichungen gelöst werden müssen, ist dadurch jedoch noch nicht gegeben. Auf Basis der zuvor bestimmten Zuordnung zwischen Gleichungen und Variablen wird ein gerichteter Graph erzeugt. Auf diesem gerichteten Graphen wird anschließend *Tarjan's Algorithmus* verwendet [Tarjan (1972)]. Das Ergebnis von Tarjan's Algorithmus ist eine Reihenfolge, in der die Gleichungen gelöst werden müssen. In einigen Fällen treten jedoch Abhängigkeiten zwischen Gleichungen untereinander auf, sodass diese nicht entkoppelt werden können, sondern zusammen gelöst werden müssen. In diesem Fall wird von einer *algebraischen Schleife* gesprochen.

Handelt es sich bei dem Gleichungssystem einer algebraischen Schleife um ein lineares Gleichungssystem, so kann dieses bei kleiner Dimension noch leicht symbolisch aufgelöst werden. Ist das Gleichungssystem jedoch nichtlinear, werden zumeist numerische Verfahren, wie z.B. das Newton-Verfahren, eingesetzt, um das Gleichungssystem iterativ zu lösen. Um den Aufwand zur Lösung des nichtlinearen Gleichungssystems zu reduzieren, wird zusätzlich ein *Tearing-Algorithmus* eingesetzt. Dieser versucht, durch eine geschickte Wahl der Iterationsvariablen die Anzahl der benötigten Variablen zu senken.

Es ist jedoch nicht in jedem Fall möglich, das DAE-System unmittelbar in ein ODE[1]-System zu überführen. Dies ist der Fall, wenn die Jacobi-Matrix

$$\frac{\partial \mathbf{F}(\mathbf{x}, \dot{\mathbf{x}}, t)}{\partial \dot{\mathbf{x}}} \tag{2.7}$$

[1] **O**rdinary **D**ifferential **E**quation

auf eine singuläre Matrix führt. Dies tritt bei einer Vielzahl technischer Systeme auf und ist in der Praxis regelmäßig der Fall. Auch in diesem Fall bleibt die Überführung in ein System gewöhnlicher Differentialgleichungen das Ziel, es muss jedoch zunächst eine *Indexreduktion* durchgeführt werden. Der (differentielle) Index[1] einer DAE ist ein Maß dafür, wie weit die DAE von einer ODE entfernt ist und gibt die minimal notwendige Anzahl der totalen Zeitableitungen der DAE an, um eine ODE zu erhalten [Gottschling & Schramm (2014)]. Eine gewöhnliche Differentialgleichung hat demnach den Index 0, während eine DAE einen Index von 1 oder höher besitzt, d.h. mindestens einmal differenziert werden muss, um eine ODE zu erhalten. Die Indexreduktion erfolgt im OpenModelica Backend mit Hilfe des *Pantelides-Algorithmus*, welcher ursprünglich ein Verfahren zur Bestimmung von konsistenten Anfangsbedingungen einer DAE darstellte. Mattsson und Söderlind haben den Algorithmus später erweitert, sodass dieser für die Indexreduktion von DAEs verwendet werden kann. Für eine detaillierte Beschreibung sei auf [Pantelides (1988)] und [Mattsson & Söderlind (1993)] verwiesen. Nach gängiger Praxis wird angenommen, dass für physikalisch sinnvolle Systeme aus den Bereichen Maschinenbau und Automatisierungstechnik die Indexreduktion stets durchgeführt werden kann, sodass die Systemgleichungen nach Durchlaufen der symbolischen Gleichungsmanipulation in Form eines Systems gewöhnlicher Differentialgleichungen vorliegen. Es muss jedoch beachtet werden, dass auch für DAEs mit geringem differentiellen Index der Aufwand für die Indexreduktion verhältnismäßig hoch sein kann [Reissig *et al.* (2000)].

2.2 Numerische Methoden zur Lösung von Differentialgleichungen

Ein objektorientiert modelliertes System liegt, wie im vorherigen Abschnitt beschrieben, nach Durchlaufen der symbolischen Gleichungsmanipulation in der so genannten Zustandsform

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \qquad \mathbf{x}(t_0) = \mathbf{x}_0. \tag{2.8}$$

vor. Derartige Systeme gewöhnlicher Differentialgleichungen können in der Regel nicht analytisch gelöst werden, sodass zur Lösung regelmäßig auf numerische Verfahren zurückgegriffen wird. Diese verfolgen im Gegensatz zur Ermittlung der exakten Lösung das Ziel, ausgehend von einem Startzeitpunkt t_0 die Lösung zum nachfolgenden Zeitpunkt $t_0 + h$ näherungsweise zu bestimmen. Der Parameter h wird als *Schrittweite* bezeichnet.

Es existiert eine Vielzahl von numerischen Integrationsverfahren, die sich allgemein in *Einschrittverfahren* sowie *Mehrschrittverfahren* unterteilen lassen. Während bei

[1] Neben dem differentiellen Indexbegriff sind weitere Index-Definitionen gebräuchlich, die hier jedoch nicht weiter verfolgt werden sollen. Für eine Übersicht sei auf [Weigl (2011)] verwiesen.

Einschrittverfahren zur Bestimmung eines Zustands $\mathbf{x}_{n+1}$ zum Zeitpunkt t_{n+1} lediglich der letzte Zustand $\mathbf{x}_n$ zum Zeitpunkt t_n verwendet wird, greifen Mehrschrittverfahren auch auf weiter zurückliegende Zustände zurück. Die im Rahmen dieser Arbeit betrachteten Verfahren entstammen der Klasse der Einschrittverfahren, die Klasse der Mehrschrittverfahren wird nicht weiter verfolgt. In den folgenden Abschnitten werden mit den Euler-Verfahren und den Runge-Kutta-Verfahren zwei bedeutende Klassen von Verfahren vorgestellt.

2.2.1 Euler-Verfahren

Die Euler-Verfahren gehören zur Klasse der Einschrittverfahren und stellen die einfachsten Verfahren zur numerischen Lösung von Differentialgleichungen dar. Nachfolgend wird sowohl das Explizite Euler-Verfahren als auch das Implizite Euler-Verfahren vorgestellt.

2.2.1.1 Explizites Euler-Verfahren

Zur Lösung des in Gleichung 2.8 dargestellten Differentialgleichungssystems wird für das Explizite Euler-Verfahren die Ableitung auf der linken Seite näherungsweise durch einen Differenzenquotienten ersetzt:

$$\dot{\mathbf{x}} = \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{h} \tag{2.9}$$

Einsetzen in Gleichung 2.8 liefert

$$\frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{h} = \mathbf{f}(\mathbf{x}_n, t_n). \tag{2.10}$$

Nach einer Umformung ergibt sich die Rechenvorschrift

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_n, t_n), \tag{2.11}$$

mit der, ausgehend von einem Startwert $\mathbf{x}_0 = \mathbf{x}(t_0)$, schrittweise die Lösung zu jedem Zeitpunkt berechnet werden kann [Gottschling & Schramm (2014)]. Dieses Verfahren stellt somit eine sehr einfache Methode zur Lösung von *Anfangswertproblemen* dar. Das Explizite Euler-Verfahren ist in Abbildung 2.2 graphisch dargestellt.

Für eine Bewertung der Güte des Verfahrens wird die exakte Lösung der Differentialgleichung in eine Taylorreihe entwickelt. Diese ist gegeben durch

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_n, t_n) + \frac{h^2}{2}\,\dot{\mathbf{f}}(\mathbf{x}_n, t_n) + \dots \tag{2.12}$$

Ein Vergleich dieser wahren Lösung mit der näherungsweise bestimmten Lösung in Gleichung 2.11 zeigt, dass der lineare Term bei beiden Lösungen übereinstimmt

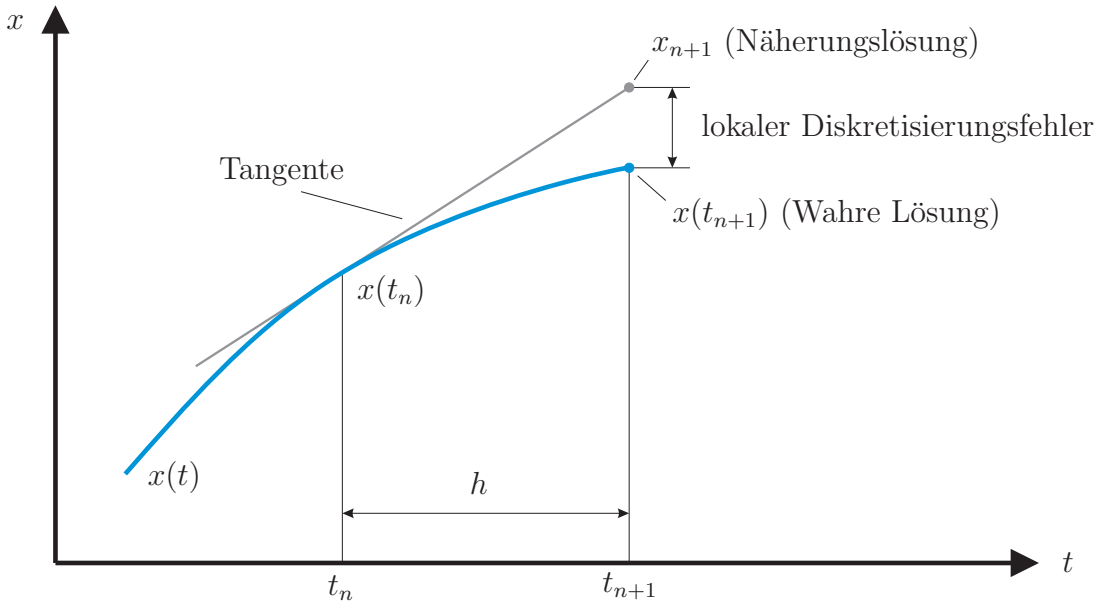

Abbildung 2.2: Numerische Integration mit dem Expliziten Euler-Verfahren [Gottschling & Schramm (2014)]

und sich die Lösungen ab dem quadratischen Term unterscheiden. Das Explizite Euler-Verfahren ist daher exakt bis zum linearen Term, die *Konsistenzordnung* des Verfahrens ist eins. Abgesehen von der niedrigen Konsistenzordnung hat das Explizite Euler-Verfahren den Nachteil, dass es nicht A-stabil ist [Gottschling & Schramm (2014)]. Die A-Stabilität ist folgendermaßen definiert.

Definition 1 (A-Stabilität)**.** Ein numerisches Verfahren ist A-stabil, wenn die komplette linke $(\lambda\, h)$-Hälfte zum Stabilitätsgebiet gehört.

Für die Untersuchung der A-Stabilität wird die *Dahlquist'sche Testgleichung*

$$\dot{x} = -\lambda\, x \tag{2.13}$$

herangezogen. Die analytische Lösung dieser Funktion, $x(t) = e^{-\lambda\, t}$, bleibt für alle komplexen Werte λ mit positivem Realteil, d.h. $\mathrm{Re}\{\lambda\} > 0$, unter Berücksichtigung des Startwertes $x_0 = x(0) > 0$ stets positiv und konvergiert für $t \to \infty$ gegen Null. Die Mindestanforderung, die an eine Näherungslösung gestellt wird, ist, dass diese ebenfalls für $t \to \infty$ gegen Null konvergiert. Es ergibt sich daraus die Bedingung

$$\frac{|x_{n+1}|}{|x_n|} < 1. \tag{2.14}$$

Für die Untersuchung des Stabilitätsverhaltens des Expliziten Euler-Verfahrens wird die Rechenvorschrift aus 2.11 auf die Testgleichung angewendet. Es ergibt sich

$$x_{n+1} = x_n - h\, \lambda\, x_n = x_n\, (1 - \lambda h). \tag{2.15}$$

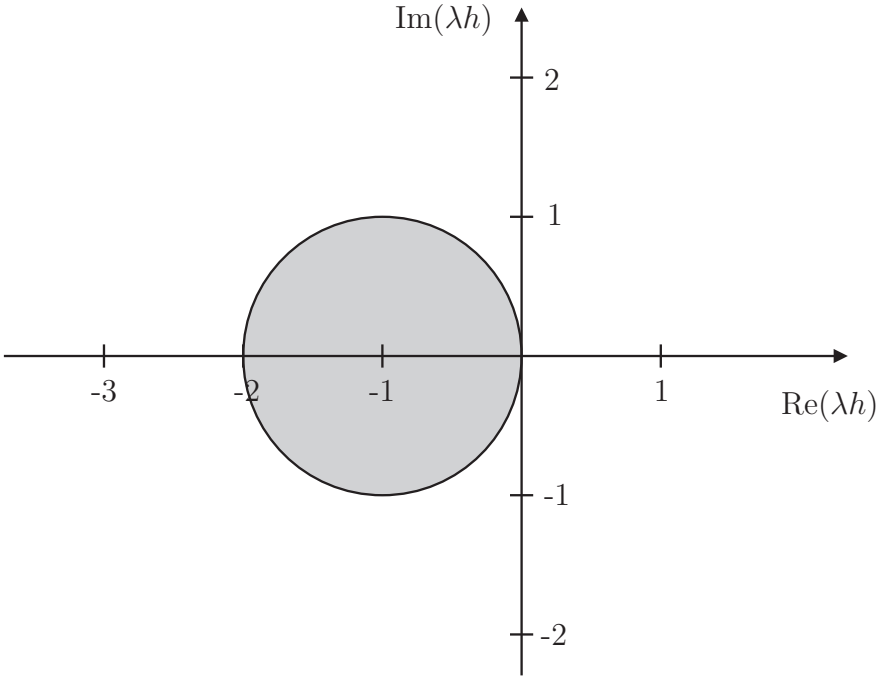

Abbildung 2.3: Stabilitätsgebiet des Expliziten Euler-Verfahrens (graue Fläche)

Diese Vorschrift erfüllt die Bedingung 2.14 genau dann, wenn $\lambda h < 1$. Die Schrittweite h muss folglich die Bedingung

$$h < \frac{1}{\lambda} \tag{2.16}$$

erfüllen [Gottschling & Schramm (2014)]. Das Verfahren ist damit nicht A-stabil. Das Stabilitätsgebiet ist ein Kreis mit dem Radius eins um den Punkt $(-1, 0)$ und ist in Abbildung 2.3 dargestellt (graue Fläche).

2.2.1.2 Implizites Euler-Verfahren

Wird der beim Expliziten Euler-Verfahren vorwärts genommene Differenzenquotient durch den rückwärts genommenen Differenzenquotienten ersetzt, ergibt sich das Implizite Euler-Verfahren. Es folgt somit

$$\frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{h} = \mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}) \tag{2.17}$$

bzw. umgeformt

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}). \tag{2.18}$$

Im Gegensatz zum Expliziten Euler-Verfahren ist in diesem Fall der Zustand $\mathbf{x}_{n+1}$ im Allgemeinen nicht mehr unmittelbar zu bestimmen, da sich, abgesehen von dem Fall,

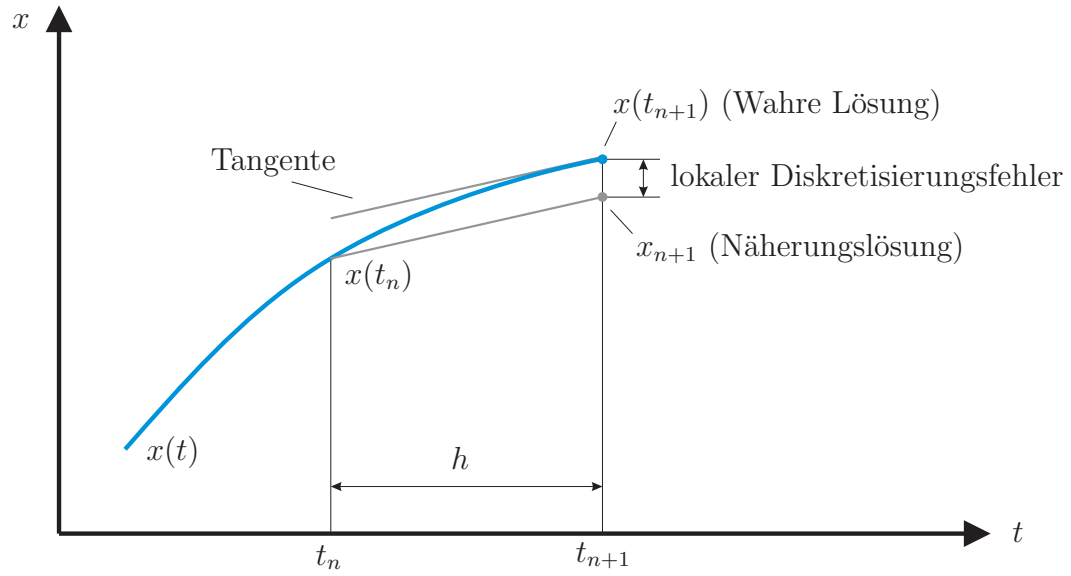

Abbildung 2.4: Numerische Integration mit dem Impliziten Euler-Verfahren [Gottschling & Schramm (2014)]

dass die Funktion $\mathbf{f}$ linear ist, die Gleichung nicht explizit nach dem unbekannten Zustand auflösen lässt. Dies führt dazu, dass ein (meist nichtlineares) Gleichungssystem gelöst werden muss. Die Bestimmung des Zustandes $\mathbf{x}_{n+1}$ erfolgt iterativ, beispielsweise mit Hilfe des Newton-Verfahrens.

Zur Bestimmung des Zustandes $\mathbf{x}_{n+1}$ wird Gleichung (2.18) umgeformt zu

$$\mathbf{x}_{n+1} - \mathbf{x}_n - h\,\mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}) = 0. \tag{2.19}$$

Wird die linke Seite als Funktion $\mathbf{F}(\mathbf{x}_{n+1})$ aufgefasst, ist das Lösen der Gleichung 2.18 gleichbedeutend mit der Aufgabe, die Nullstelle von $\mathbf{F}$ zu bestimmen.

Newton-Verfahren Das Newton-Verfahren ist ein numerisches Verfahren zum Lösen nichtlinearer Gleichungssysteme, was gleichbedeutend mit einer Nullstellenbestimmung ist. Idee ist es, ausgehend von einem Startpunkt $\mathbf{x}^{(0)}$ an eben diesem Punkt eine Tangente an die Funktion zu legen. Die Nullstelle der Tangente wird dann als erste Näherungslösung $\mathbf{x}^{(1)}$ für die Nullstelle der Funktion verwendet. Nun wird an diesem Punkt ebenfalls die Tangente an die Funktion gelegt und das Verfahren so lange fortgeführt, bis ein Abbruchkriterium erfüllt ist.

Die Rekursionsvorschrift für das Newton-Verfahren ist für den Fall einer eindimensionalen Funktion $\mathrm{f} : \mathbb{R} \to \mathbb{R}$

$$x^{(l+1)} = x^{(l)} - \frac{\mathrm{f}(x^{(l)})}{\mathrm{f}'(x^{(l)})} \tag{2.20}$$

sowie für den Fall einer mehrdimensionalen Funktion $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^m$ nach Ersetzen der Ableitung aus (2.20) durch die Jacobi-Matrix $\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$

$$\mathbf{x}^{(l+1)} = \mathbf{x}^{(l)} - \mathbf{J}^{-1}(\mathbf{x}^{(l)})\,\mathbf{f}(\mathbf{x}^{(l)}). \tag{2.21}$$

Das Newton-Verfahren ist auf die Funktion $\mathbf{F}$ anzuwenden. Die Jacobi-Matrix $\mathbf{T}$ der Funktion $\mathbf{F}(\mathbf{x}_{n+1})$ ist per Definition

$$\mathbf{T} = \frac{\partial \mathbf{F}(\mathbf{x}_{n+1})}{\partial \mathbf{x}_{n+1}} = \mathbf{I} - h\,\frac{\partial \mathbf{f}(\mathbf{x}_{n+1})}{\partial \mathbf{x}_{n+1}} = \mathbf{I} - h\,\mathbf{J}, \tag{2.22}$$

wobei $\mathbf{I}$ die Einheitsmatrix der Dimension $m \times m$ ist. Als Iterationsvorschrift für die unbekannte Lösung $\mathbf{x}_{n+1}$ ergibt sich damit analog zu (2.21)

$$\mathbf{x}_{n+1}^{(l+1)} = \mathbf{x}_{n+1}^{(l)} - \mathbf{T}^{-1}(\mathbf{x}_{n+1}^{(l)})\,\mathbf{F}(\mathbf{x}_{n+1}^{(l)}). \tag{2.23}$$

Als Startwert der Iteration $\mathbf{x}_{n+1}^{(0)}$ wird die alte Lösung $\mathbf{x}_n$ verwendet.

Das Implizite Euler-Verfahren ist in Abbildung 2.4 graphisch dargestellt. Genau wie das Explizite Euler-Verfahren besitzt auch das Implizite Euler-Verfahren die Konsistenzordnung eins [Cellier & Kofman (2006)]. Es ist jedoch A-stabil. Das Stabilitätsgebiet ist in Abbildung 2.5 abgebildet, es handelt sich um die weiße Fläche. Es ist zu erkennen, dass insbesondere die komplette linke $(\lambda\, h)$-Hälfte zum Stabilitätsgebiet gehört.

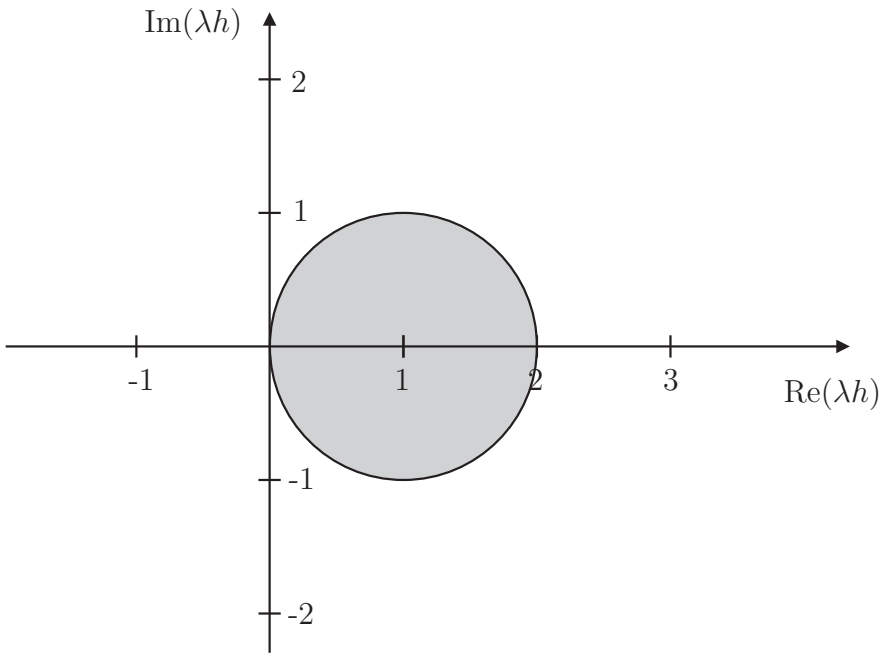

Abbildung 2.5: Stabilitätsgebiet des Impliziten Euler-Verfahrens (weiße Fläche)

2.2.2 Runge-Kutta-Verfahren

Aufgrund der niedrigen Konsistenzordnung eignen sich die Euler-Verfahren nur bedingt für Simulationen, da sie aufgrund der nicht sehr hohen Genauigkeit und der dadurch bedingten geringen Schrittweite für einen hohen Rechenaufwand sorgen. Daher gab es schon recht früh Ansätze, ein Lösungsverfahren höherer Ordnung zu entwickeln. Dies führt zur Klasse der Runge-Kutta-Verfahren. Um die Genauigkeit weiter zu verbessern, sind weitere Funktionsauswertungen innerhalb des Intervalls $[t_n; t_{n+1}]$ notwendig. Im Folgenden werden das Explizite Runge-Kutta-Verfahren und das Implizite Runge-Kutta-Verfahren vorgestellt.

2.2.2.1 Explizites Runge-Kutta-Verfahren

Das erste Runge-Kutta-Verfahren wurde 1895 von Carl Runge vorgestellt [Runge (1895)]. Allgemein lauten die Berechnungsvorschriften für ein s-stufiges Explizites Runge-Kutta-Verfahren:

$$\mathbf{k}_i = \mathbf{f}\left(\mathbf{x}_n + h \sum_{j=1}^{i-1} a_{ij}\, \mathbf{k}_j,\ t_n + c_i\, h\right), \quad 1 \leq i \leq s \tag{2.24}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \sum_{i=1}^{s} b_i\, \mathbf{k}_i, \tag{2.25}$$

wobei $a_{ij}, b_i, c_i \in \mathbb{R}$ Konstanten sind und $s \in \mathbb{R}$ die Anzahl der Stufen bezeichnet. Für die Berechnung eines Integrationsschrittes werden somit bei Verwendung eines Verfahrens der Ordnung s genau s Funktionsauswertungen benötigt.

Die Konsistenzordnung p eines Expliziten Runge-Kutta-Verfahrens ist mit der Anzahl der Stufen s beschränkt, d.h. es existiert kein Verfahren mit $p > s$. Für $p \geq 5$ gilt darüber hinaus die sogenannte Butcher-Barriere [Hairer *et al.* (1993)]. Diese besagt, dass kein Explizites Runge-Kutta-Verfahren mit $p = s$ existiert. Es existieren jedoch beispielsweise sechsstufige Verfahren mit Konsistenzordnung fünf oder auch 11-stufige Verfahren der Konsistenzordnung acht [Cellier & Kofman (2006)].

Die Koeffizienten der Runge-Kutta-Verfahren werden in *Butcher-Tableaus* angeordnet. Diese besitzen die Form

$\mathbf{c}$	$\mathbf{A}$
x	$\mathbf{b}^T$

mit

$$\mathbf{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_s \end{bmatrix} \quad , \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_s \end{bmatrix} \quad , \quad \mathbf{A} = \begin{bmatrix} a_{11} & \dots & a_{1m} & \dots & a_{1s} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nm} & \dots & a_{ns} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{s1} & \dots & a_{sm} & \dots & a_{ss} \end{bmatrix} . \tag{2.26}$$

Das in Abschnitt 2.2.1.1 vorgestellte Explizite Euler-Verfahren lässt sich als das einfachste Explizite Runge-Kutta-Verfahren auffassen. Das zugehörige Butcher-Tableau lautet

0	0
x	1

Tabelle 2.2: Butcher-Tableau: Explizites Euler-Verfahren [Gottschling & Schramm (2014)]

Das am häufigsten verwendete, klassische Runge-Kutta-Verfahren (RK4) besitzt die Konsistenzordnung vier und wird durch das folgende Butcher-Tableau beschrieben:

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	0	0
1	0	0	1	0
x	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

Tabelle 2.3: Butcher-Tableau: RK4 [Gottschling & Schramm (2014)]

Im Gegensatz zu dem Expliziten Euler-Verfahren sind innerhalb der Klasse der Runge-Kutta-Verfahren Verfahren mit einer höheren Konsistenzordnung zu finden. Jedoch ist keines dieser Verfahren A-stabil [Hairer *et al.* (1993)]. Die Stabilitätsgebiete der Runge-Kutta-Verfahren der Ordnungen eins bis vier sind in Abbildung 2.6 dargestellt.

2.2.2.2 Implizites Runge-Kutta-Verfahren

Die Berechnungsvorschrift für ein s-stufiges Implizites Runge-Kutta-Verfahren lautet allgemein

$$\mathbf{k}_i = \mathbf{f}\left(\mathbf{x}_n + h \sum_{j=1}^{s} a_{ij}\, \mathbf{k}_j,\ t_n + c_i\, h\right), \quad 1 \leq i \leq s \tag{2.27}$$

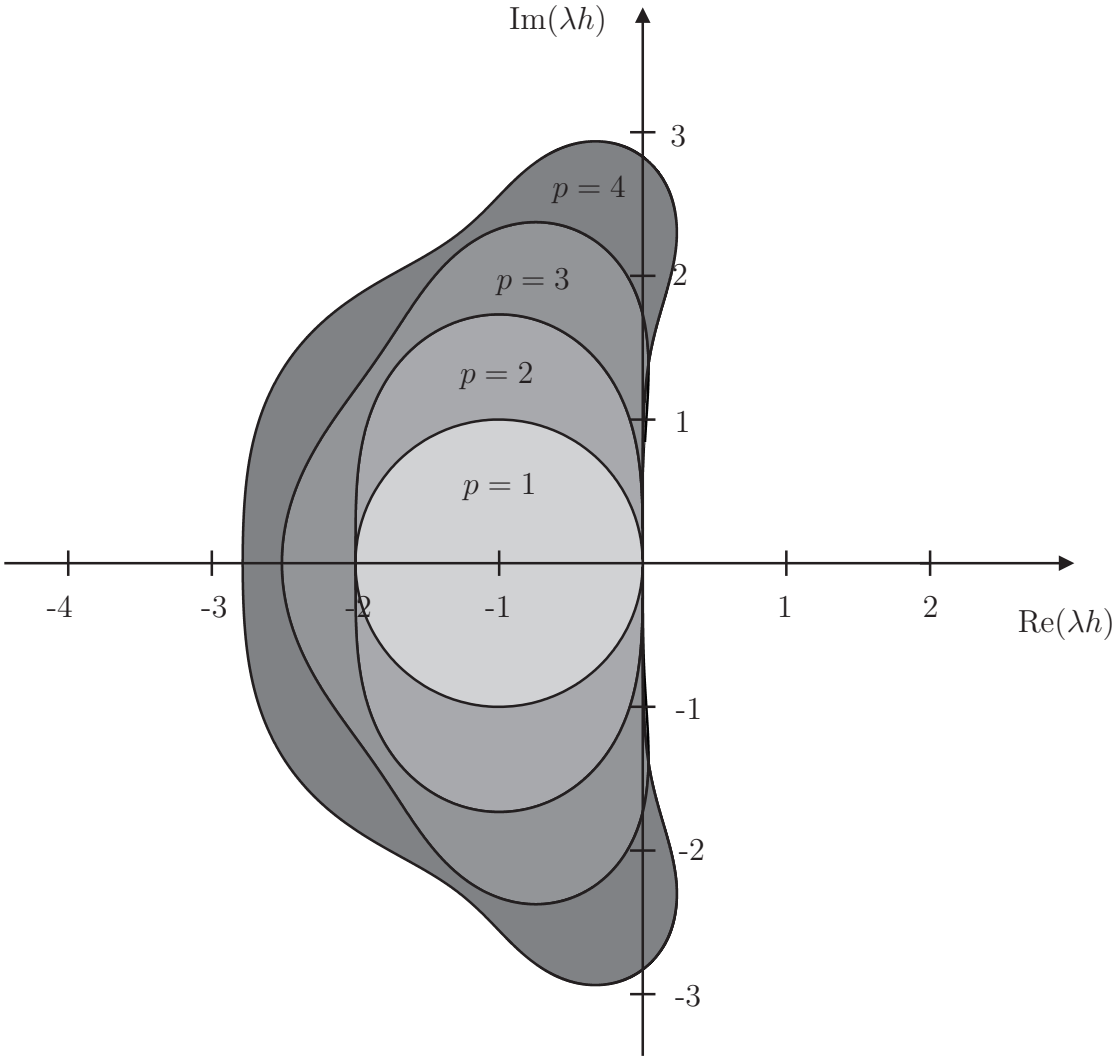

Abbildung 2.6: Stabilitätsgebiete der Expliziten Runge-Kutta-Verfahren der Ordnungen eins bis vier

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \sum_{i=1}^{s} b_i \, \mathbf{k}_i. \tag{2.28}$$

Ob ein Verfahren explizit oder implizit ist, lässt sich ebenfalls sehr leicht an dem Butcher-Tableau erkennen. Ist die Matrix $\mathbf{A}$ eine strikte untere Dreiecksmatrix, d.h. $a_{ij} = 0$ für alle $i \leq j$, so ist das Verfahren explizit, andernfalls implizit. Gilt $a_{ij} = 0$ für alle $i < j$ und ist mindestens ein $a_{ii} \neq 0$, so wird von einem Diagonal-Impliziten Runge-Kutta-Verfahren gesprochen. Die Klasse der Impliziten Runge-Kutta-Verfahren enthält eine Vielzahl stabiler Lösungsverfahren auch für steife Systeme. Viele der häufig eingesetzten Verfahren im Rahmen der Offline-Simulation entstammen dieser Klasse, wie beispielsweise das RadauIIa-Verfahren.

2.3 Lösung mathematischer Optimierungsprobleme

Mit dem Begriff der Optimierung wird allgemein die Suche einer in einem bestimmten Sinne optimalen Lösung für ein definiertes Problem bezeichnet. Für die Bewertung der Güte der Lösung wird eine *Kostenfunktion* herangezogen, die - je nach Aufgabenstellung - entweder einen minimalen (Minimierungsproblem) oder einen maximalen (Maximierungsproblem) Funktionswert aufweisen soll. Optimierungsprobleme treten in den verschiedensten Fachdisziplinen auf. So kann im Bereich der Betriebswirtschaftslehre beispielsweise untersucht werden, welche Produktionsmenge eines Gutes bei gegebener Kostenfunktion gewinnmaximal ist. Im Bereich der Robotik ist es unter anderem wünschenswert, eine für einen Roboter *optimale* Bahn zu bestimmen. Die optimale Lösung kann dabei zum Beispiel unter Berücksichtigung des Ziels der Energieminimierung oder der Zeitminimierung bestimmt werden.

Auch das bereits in der Motivation genannte Problem, unter Berücksichtigung eines vorgegebenen Sollverhaltens für ein technisches System eine optimale Kombination aus verwendeten Systemkomponenten und Reglerparametern zu bestimmen, lässt sich mathematisch als Optimierungsproblem beschreiben. Auf dieses Optimierungsproblem wird in Kapitel 3.2 genauer eingegangen. Im Rahmen dieser Arbeit werden ausschließlich Minimierungsprobleme betrachtet. Die Behandlung von Minimierungs- und Maximierungsproblemen erfolgt mit den selben Methoden, da ohnehin jedes Minimierungsproblem durch Multiplikation der Kostenfunktion mit -1 in ein Maximierungsproblem umgewandelt werden kann und umgekehrt.

Klassifizierung von Optimierungsproblemen Im Allgemeinen wird innerhalb der Gruppe der Optimierungsprobleme zwischen *statischen* und *dynamischen* Optimierungsproblemen unterschieden. Die Lösung eines statischen Optimierungsproblems beinhaltet die Minimierung einer Funktion mit Optimierungsvariablen $\mathbf{x}$, die Elemente des euklidischen Raumes $\mathbb{R}^n$ sind. Auf der anderen Seite beinhaltet die Lösung eines dynamischen Optimierungsproblems die Minimierung eines Funktionals zur Bestimmung einer Lösungsfunktion $\mathbf{x}(t)$ mit einer unabhängigen Variablen t. Die Optimierungsvariablen sind in diesem Fall Elemente eines allgemeineren Hilbertraumes [Papageorgiou (2012)]. Die unabhängige Variable ist in der Regel die Zeit t, sie kann jedoch im Prinzip eine beliebige Bedeutung haben. Wird im Rahmen dieser Arbeit von Optimierungsproblemen gesprochen, so sind stets statische Probleme gemeint.

Derartige statische Optimierungsprobleme lassen sich allgemein in der Form

$$\min_{\mathbf{x}\in\mathbb{R}^n} \quad f(\mathbf{x}) \tag{2.29}$$

$$\text{u.d.N.}^1 \quad g_i(\mathbf{x}) = 0, \quad i = 1, \ldots, p \tag{2.30}$$

$$h_i(\mathbf{x}) \leq 0, \quad i = 1, \ldots, q \tag{2.31}$$

[1] unter den Nebenbedingungen

beschreiben [Graichen (2012)]. Dabei wird die Funktion $f(\mathbf{x})$ als Kostenfunktion bezeichnet. Innerhalb dieses Kapitels werden lediglich Optimierungsprobleme betrachtet, die *ein* Optimierungsziel besitzen. Für diese Probleme ist die Kostenfunktion $f(\mathbf{x})$ skalar. Die Minimierung der Kostenfunktion erfolgt in der Regel unter Berücksichtigung von Nebenbedingungen. Die Nebenbedingungen können sowohl in Form einer Gleichung als auch in Form einer Ungleichung vorliegen. Wird ein Optimierungsproblem ohne Nebenbedingungen formuliert, wird dieses auch als *unbeschränktes* Optimierungsproblem bezeichnet [Kallrath (2013)].

Zur näheren Beschreibung eines Optimums werden zwei Definitionen benötigt. Für eine Funktion $f(\mathbf{x})$, die auf dem Gebiet $S \subseteq \mathbb{R}^n$ definiert ist, ist das Minimum (Optimum) folgendermaßen definiert:

Definition 2 (Globales Minimum)**.** Die Funktion f besitzt ein (globales) Minimum an der Stelle $\bar{\mathbf{x}} \in S$, falls gilt

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) \; \forall \, \mathbf{x} \in S.$$

Definition 3 (Lokales Minimum)**.** Die Funktion f besitzt ein lokales Minimum an der Stelle $\bar{\mathbf{x}} \in S$, falls eine Umgebung von $\bar{\mathbf{x}}$, $\Omega = \{\mathbf{x} \in S \, : \, \|\mathbf{x} - \bar{\mathbf{x}}\| < \delta, \delta > 0\}$, existiert, sodass gilt

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) \; \forall \, \mathbf{x} \in \Omega.$$

Abhängig von der Beschaffenheit der Kostenfunktion können die statischen Optimierungsprobleme weiter klassifiziert werden. Handelt es sich bei der Funktion $f(\mathbf{x})$ um eine lineare Funktion, wird das Problem als *lineares Optimierungsproblem* bezeichnet (Linear Program, LP). Ist die Funktion nichtlinear, wird von einem *nichtlinearen Optimierungsproblem* gesprochen (Nonlinear Program, NLP). Während bei linearen Optimierungsproblemen nur eine Formulierung unter Berücksichtigung von Nebenbedingungen sinnvoll ist, da ansonsten kein Optimum existiert, existieren nichtlineare Optimierungsprobleme sowohl mit als auch ohne Berücksichtigung von Nebenbedingungen.

Wird für die Optimierungsvariablen zusätzlich festgelegt, dass diese nur ganzzahlig sein dürfen, ergibt sich die Klasse der *ganzzahligen Optimierungsprobleme*. Ist ein Teil der Optimierungsvariablen reellwertig und ein Teil ganzzahlig, wird von *gemischt-ganzzahligen Optimierungsproblemen* gesprochen. Auch bei dieser Klasse wird zwischen *linearen gemischt-ganzzahligen Optimierungsproblemen* (Mixed-Integer Linear Program, MILP) und *nichtlinearen gemischt-ganzzahligen Optimierungsproblemen* (Mixed-Integer Nonlinear Program, MINLP) unterschieden. Die Klassifizierungsmöglichkeiten sind in Abbildung 2.7 dargestellt. Der Vollständigkeit

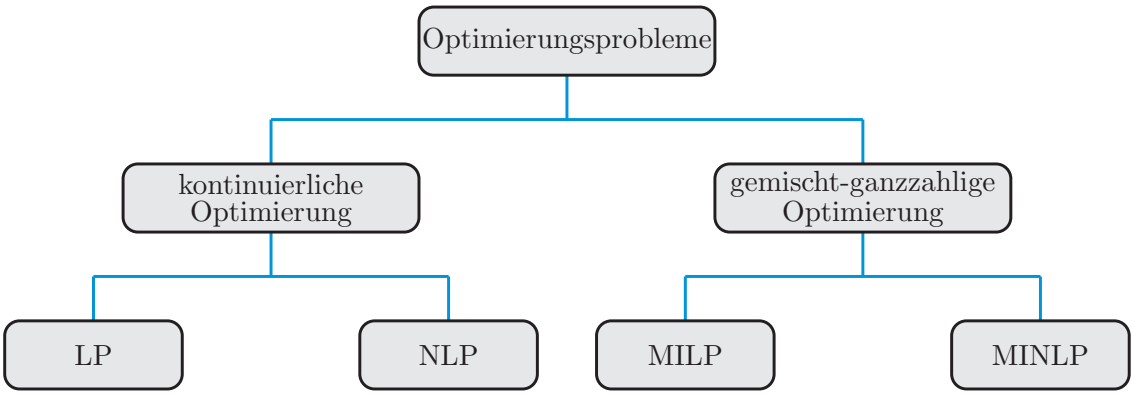

Abbildung 2.7: Klassifizierung der wichtigsten statischen Optimierungsprobleme

halber sei erwähnt, dass neben den zuvor genannten weitere Klassen existieren, beispielsweise die Klasse der gemischt-ganzzahligen quadratischen Optimierungsprobleme (Mixed-Integer Quadratic Program, MIQP). Diese besitzen eine quadratische, konvexe Kostenfunktion. Da diese für die in dieser Arbeit betrachteten Probleme jedoch keine Rolle spielen, werden diese nicht weiter betrachtet. Die Lösung derartiger Optimierungsprobleme wird unter anderem in [Lazimy (1982)] und [Gill & Wong (2015)] betrachtet.

Für die Lösung der in der Motivation beschriebenen Optimierungsprobleme, deren Lösung die optimale Kombination aus Reglerparametern sowie Systemkomponenten ist, werden Methoden für nichtlineare gemischt-ganzzahlige Optimierungsprobleme (MINLP) benötigt. Im Rahmen der Parameteridentifikation bei der Modellbildung sind die Optimierungsvariablen in der Regel rein reellwertig, weswegen Methoden zur Lösung von nichtlinearen Optimierungsproblemen (NLP) ausreichend sind. In diesem Kontext wird in Abschnitt 2.3.1.1 das Gradientenverfahren sowie in Abschnitt 2.3.1.2 das Nelder-Mead-Verfahren behandelt. Die in Abschnitt 2.3.2 beschriebenen Verfahren der Klasse der naturanalogen Optimierungsmethoden sind zusätzlich in der Lage, gemischt-ganzzahlige Optimierungsprobleme zu lösen. Darüber hinaus bieten diese Verfahren eine bessere Chance, das globale Optimum zu finden. Das Auffinden globaler Optima ist bei den gradientenbasierten Verfahren, zu welchen sowohl das Gradientenverfahren als auch das Nelder-Mead-Verfahren gezählt werden können[1], grundsätzlich problematisch, da diese häufig nur lokale Optima finden bzw. das Optimierungsergebnis stark von der Wahl des Startpunktes abhängt.

[1] Obwohl das Nelder-Mead-Verfahren kein gradientenbasiertes Verfahren im klassischen Sinne ist, da der Gradient nicht explizit berechnet wird, kann es dennoch den (quasi-)gradientenbasierten Verfahren zugerechnet werden, da der Gradient innerhalb des Algorithmus angenähert wird.

2.3.1 Gradientenbasierte Verfahren

Gradientenbasierte Verfahren lassen sich immer dann gut einsetzen, wenn die Optimierungsvariablen reellwertig sind und die Kostenfunktion differenzierbar ist. Die Klasse von Verfahren zeichnet sich durch die hohe Konvergenzgeschwindigkeit aus, insbesondere dann, wenn der Gradient einfach berechnet werden kann. Beispiele sind das *Gradientenverfahren* (Methode des steilsten Abstiegs) und das *Newton-* bzw. das *Quasi-Newton-Verfahren* [Geiger & Kanzow (1999)]. Das Gradientenverfahren benötigt dazu den Gradienten der Kostenfunktion, während das Newton-Verfahren zusätzlich zum Gradienten ebenfalls die Hesse-Matrix benötigt.

Gerade für mehrdimensionale Optimierungsprobleme ist die Berechnung der Ableitungen jedoch häufig mit einem hohem Rechenaufwand verbunden. Für diese Art von Problemen existieren spezielle *ableitungsfreie* Optimierungsverfahren. Ein wichtiger Vertreter ist das *Nelder-Mead-Verfahren*, welches in Abschnitt 2.3.1.2 beschrieben wird.

2.3.1.1 Gradientenverfahren

Das Gradientenverfahren, auch als Verfahren des steilsten Abstiegs bekannt, ist das einfachste Verfahren zum Lösen von kontinuierlichen Optimierungsproblemen ohne Nebenbedingungen. Ausgehend von einem Startpunkt $\mathbf{x}^{(0)}$ wird mit einer definierten, aber variablen Schrittweite α so lange ein Schritt in Richtung des negativen Gradienten

$$\mathbf{d} = -\nabla f(\mathbf{x}^{(l)}) \tag{2.32}$$

der Funktion durchgeführt, bis ein Abbruchkriterium erfüllt ist. Als Abbruchkriterium wird häufig die Differenz zweier aufeinander folgender Näherungslösungen $\mathbf{x}^{(l)}$ und $\mathbf{x}^{(l+1)}$ betrachtet. Der Algorithmus stoppt, sobald die Differenz kleiner als eine vorher definierte Schranke ϵ ist. Die Schrittweite α wird in jedem Iterationsschritt neu berechnet. Für die Iterationsvorschrift des Verfahrens gilt

$$\mathbf{x}^{(l+1)} = \mathbf{x}^{(l)} - \alpha^{(l)} \cdot \nabla f(\mathbf{x}^{(l)}). \tag{2.33}$$

Für die Berechnung der Schrittweite stehen verschiedene Ansätze zur Verfügung. Nach [Yuan (2008)] besteht eine Möglichkeit darin, die Schrittweite $\alpha^{(l)}$ durch Lösen der notwendigen Optimalitätsbedingung (Nullsetzen der Ableitung) für die betrachtete Iteration zu bestimmen. Mathematisch wird dies mit Hilfe der Formel

$$\alpha^{(l)} = \min_{\alpha>0}\{\alpha \mid \nabla f(\mathbf{x}^{(l)} - \alpha \cdot \mathbf{g}^{(l)}) = 0\} \tag{2.34}$$

beschrieben, wobei gilt

$$\mathbf{g}^{(l)} = \nabla f(\mathbf{x}^{(l)}). \tag{2.35}$$

Das Verfahren ist einfach zu implementieren und zeigt anschaulich, wie ausgehend von einem Startpunkt das Optimum gefunden wird. Jedoch handelt es sich bei dem Gradientenverfahren um eine Methode zum Auffinden lokaler Optima. Es gibt keine Möglichkeit zu überprüfen, ob es sich bei der gefundenen Lösung um ein lokales oder um das globale Optimum handelt. Ferner hängt die gefundene Lösung bei einer Zielfunktion mit vielen lokalen Optima stark von dem gewählten Startpunkt $\mathbf{x}^{(0)}$ ab. Die Konvergenzgeschwindigkeit des Verfahrens nimmt mit steigender Annäherung an das Optimum ab, da sowohl die berechnete Schrittweite als auch der Betrag des Gradienten stets kleiner werden.

2.3.1.2 Nelder-Mead-Verfahren

Das Nelder-Mead-Verfahren, auch als Downhill-Simplex-Verfahren bezeichnet, ist ein ableitungsfreies Optimierungsverfahren zur Lösung von nichtlinearen Optimierungsproblemen ohne Nebenbedingungen. Das Verfahren wurde erstmals im Jahr 1965 von J. Nelder und R. Mead vorgestellt [Nelder & Mead (1965)]. Es ist zu beachten, dass das Nelder-Mead-Verfahren nicht mit dem zum Lösen von linearen Optimierungsproblemen verwendeten Simplex-Verfahren verwechselt werden darf, welches ebenfalls auf einem Simplex basiert.

Ein Simplex ist das einfachste Polytop, eine Verallgemeinerung des Polygons in beliebige Dimensionen, welches durch jeden weiteren Punkt um eine neue Dimension erweitert wird. Es wird im $\mathbb{R}^n$ beschrieben durch $n+1$ Ecken $\mathbf{x}_1, \ldots, \mathbf{x}_{n+1} \in \mathbb{R}^n$, sodass die Vektoren $\mathbf{x}_i - \mathbf{x}_1$ für $i = 2, \ldots, n+1$ linear unabhängig sind. Das Simplex der Dimension k wird häufig auch als k-Simplex bezeichnet. In diesem Sinn ist das 0-Simplex ein Punkt, das 1-Simplex eine Gerade, das 2-Simplex ein Dreieck usw. [Lee (2010)].

In jedem Iterationsschritt des Nelder-Mead-Verfahrens wird ein neues Simplex erzeugt. Dabei wird immer die Ecke des Simplex, die den schlechtesten Funktionswert besitzt, durch einen „besseren Punkt“ ersetzt. Hierzu werden zu Beginn eines jeden Iterationsschrittes die Funktionswerte aller Ecken in aufsteigender Reihenfolge angeordnet, sodass gilt

$$f(\mathbf{x}_1) \leq \cdots \leq f(\mathbf{x}_{n+1}). \tag{2.36}$$

Der Punkt $\mathbf{x}_1$ besitzt folglich den besten (d.h. niedrigsten) Funktionswert, während der Punkt $\mathbf{x}_{n+1}$ den höchsten Funktionswert aufweist. Um für die „schlechteste“ Ecke einen besseren Punkt zu erhalten, wird der neue Punkt auf der Geraden durch diesen Punkt $\mathbf{x}_{n+1}$ und den Schwerpunkt der übrigen Ecken

$$\bar{\mathbf{x}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i \tag{2.37}$$

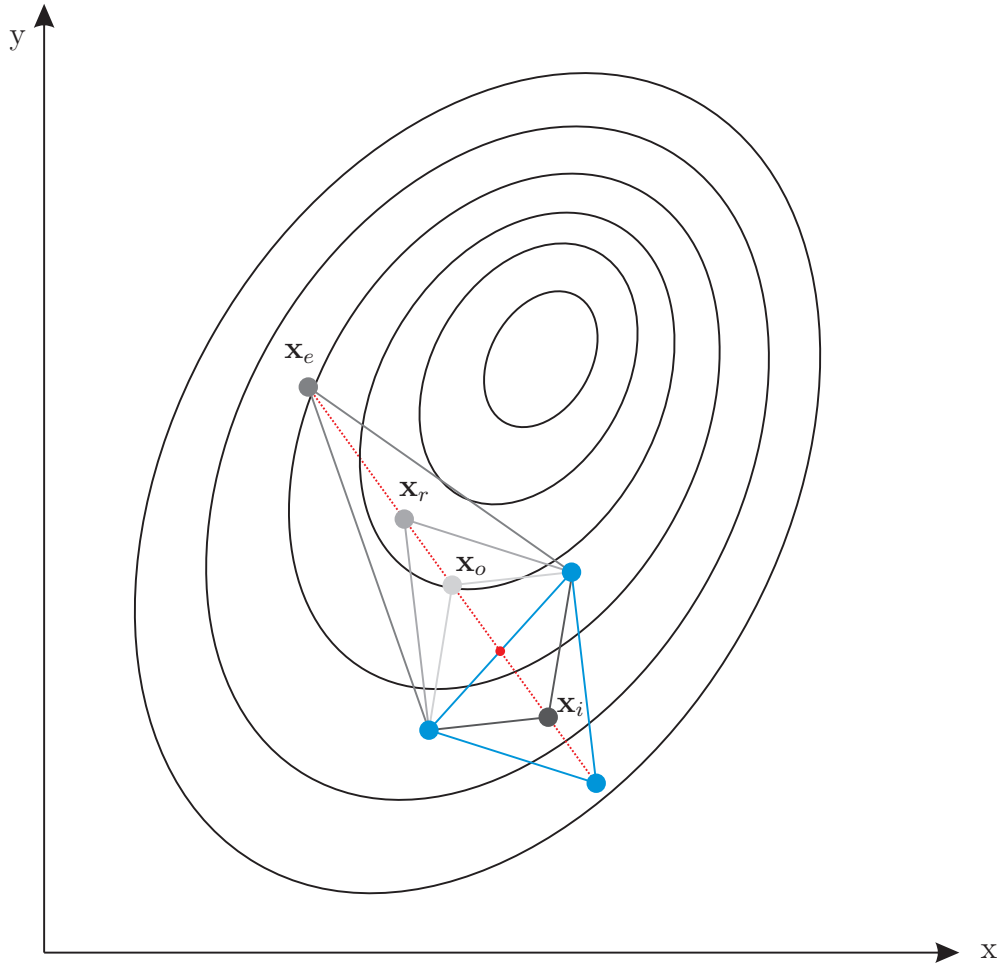

Abbildung 2.8: Operatoren des Nelder-Mead-Verfahrens ausgehend von dem in blau dargestellten Startsimplex (Verwendung der Standardparametrierung aus [Nelder & Mead (1965)])

gesucht. Dafür können innerhalb des Algorithmus vier Operationen durchgeführt werden, mit denen die Position des neuen Punktes auf der Geraden festgelegt wird. Diese Operationen lauten

- Reflexion (reflection),
- Expansion (expansion),
- Kontraktion (outside contraction) sowie
- Komprimierung (inside contraction).

Die Auswirkung der möglichen Operatoren ist in Abbildung 2.8 dargestellt. Im ersten Schritt wird eine Reflexion durchgeführt. Es ergibt sich der Punkt $\mathbf{x}_r$. Abhängig von dem Funktionswert des reflektierten Punktes im Bezug auf die Funktionswerte

Algorithmus 1 Nelder-Mead Verfahren

1. **Sortieren**. Werte f an den $n+1$ Ecken des Simplex aus und ordne die Funktionswerte entsprechend Gleichung 2.36 an

2. **Reflexion**. Berechne den Reflexionspunkt x_r durch

$$\mathbf{x}_r = \bar{\mathbf{x}} + \alpha(\bar{\mathbf{x}} - \mathbf{x}_{n+1})$$

   Berechne $f_r = f(\mathbf{x}_r)$. Falls $f_1 < f_r < f_n$, ersetze $\mathbf{x}_{n+1}$ durch $\mathbf{x}_r$.

3. **Expansion**. Falls $f_r < f_1$ berechne den Expansionspunkt $\mathbf{x}_e$ durch

$$\mathbf{x}_e = \bar{\mathbf{x}} + \beta(\mathbf{x}_r - \bar{\mathbf{x}})$$

   und berechne $f_e = f(\mathbf{x}_e)$. Falls $f_e < f_r$, ersetze $\mathbf{x}_{n+1}$ durch $\mathbf{x}_e$; ansonsten ersetze $\mathbf{x}_{n+1}$ durch $\mathbf{x}_r$.

4. **Kontraktion**. Falls $f_n \leq f_r < f_{n+1}$, berechne den Kontraktionspunkt

$$\mathbf{x}_{oc} = \bar{\mathbf{x}} + \gamma(\mathbf{x}_r - \bar{\mathbf{x}})$$

   und berechne $f_{oc} = f(\mathbf{x}_{oc})$. Falls $f_{oc} \leq f_r$, ersetze $\mathbf{x}_{n+1}$ durch $\mathbf{x}_{oc}$; ansonsten gehe zu Schritt 6.

5. **Komprimierung**. Falls $f_r \geq f_{n+1}$, berechne den Komprimierungspunkt $\mathbf{x}_{ic}$ durch

$$\mathbf{x}_{ic} = \bar{\mathbf{x}} - \gamma(\mathbf{x}_r - \bar{\mathbf{x}})$$

   und berechne $f_{ic} = f(\mathbf{x}_{ic})$. Falls $f_{ic} < f_{n+1}$, ersetze $\mathbf{x}_{n+1}$ durch $\mathbf{x}_{ic}$; ansonsten gehe zu Schritt 6.

6. **Schrumpfen**. Für $2 \leq i \leq n+1$, setze

$$\mathbf{x}_i = \mathbf{x}_1 + \delta(\mathbf{x}_i - \mathbf{x}_1).$$

der übrigen Ecken wird entweder die zu verbessernde Ecke sofort durch den neu berechneten Reflexionspunkt ersetzt oder eine weitere Operation (Expansion, Kontraktion, Komprimierung) ausgeführt. Für den Fall der Expansion ergibt sich der Punkt $\mathbf{x}_e$, für die Kontraktion der Punkt $\mathbf{x}_o$ und für den Fall der Komprimierung ergibt sich der Punkt $\mathbf{x}_i$. Der komplette Algorithmus ist in Algorithmus 1 erläutert und ist aus [Nelder & Mead (1965)] entnommen. Dieser Algorithmus wird solange durchgeführt, bis ein Abbruchkriterium erfüllt ist.

Für die im Algorithmus eingeführten Parameter gelten folgende Restriktionen: $\alpha > 0$, $\beta > 1$, $0 < \gamma < 1$ und $0 < \delta < 1$. Die Standardparametrierung des Nelder-Mead-Verfahrens verwendet die Konfiguration [Nelder & Mead (1965)]

$$\{\alpha,\ \beta,\ \gamma,\ \delta\} = \{1,\ 2,\ \frac{1}{2},\ \frac{1}{2}\}. \tag{2.38}$$

Es werden im Wesentlichen drei unterschiedliche Abbruchkriterien verwendet:

- Konvergenzkriterium für x: Das Simplex ist *genügend klein*,
- Konvergenzkriterium für y: Die Funktionswerte der Simplex-Ecken sind *genügend nah beieinander*,
- Abbruchkriterium ohne Konvergenz: Maximale Anzahl an Iterationen oder Funktionsauswertungen erreicht.

Ein Beispiel, welches die Funktionsweise des Nelder-Mead-Verfahrens anschaulich zeigt, findet sich in [Menager (2012)]. Obwohl das Nelder-Mead-Verfahren ein Verfahren für Optimierungsprobleme ohne Nebenbedingungen ist, gibt es Ansätze, wie Nebenbedingungen eingebaut werden können [Gedda (2011)]. Dies geschieht im Wesentlichen durch eine Modifikation der Kostenfunktion dahingehend, dass eine weitere Funktion additiv hinzugefügt wird, die die Nebenbedingung enthält. Hierfür kann eine *Penalty-Funktion* oder eine *Barrier-Funktion* verwendet werden.

Die Penalty-Funktion sorgt dafür, dass der Kostenfunktionswert stark ansteigt, wenn der zulässige Bereich *verlassen* wird. Nichtzulässige Punkte sind also weiterhin erlaubt, sorgen jedoch für einen „Strafwert“ innerhalb der Kostenfunktion. Dieser Ansatz ist daher nicht einsetzbar, wenn die Nebenbedingung strikt eingehalten werden muss. In solchen Fällen eignet sich der Einsatz einer Barrier-Funktion, die aufgrund ihrer Beschaffenheit dafür sorgt, dass der Kostenfunktionswert unendlich groß wird, sobald der Rand des zulässigen Gebietes *erreicht* wird.

2.3.2 Naturanaloge Optimierungsverfahren

Neben den *deterministischen Optimierungsverfahren*, zu denen die im vorherigen Abschnitt vorgestellten Verfahren gehören, können *Metaheuristiken* verwendet werden. Metaheuristiken sind Algorithmen zur näherungsweisen Lösung von Optimierungsproblemen. Während bei deterministischen Verfahren der Algorithmus garantiert zu einem Optimum führt, wobei dieses Optimum ebenfalls ein lokales Optimum sein kann, kann dies bei Metaheuristiken nicht garantiert werden. Da diese Optimierungsverfahren zufallsbasierte Elemente enthalten, werden die Methoden zu den *stochastischen Optimierungsverfahren* gezählt. Metaheuristiken werden häufig dann eingesetzt, wenn kein effizienter Lösungsalgorithmus bekannt ist, wie dies beispielsweise bei nichtlinearen gemischt-ganzzahligen Optimierungsproblemen der Fall ist.

Durch Nutzung derartiger Verfahren ist es möglich, bereits innerhalb kurzer Zeit hinreichend gute Lösungen zu erhalten.

Naturanaloge Verfahren sind Metaheuristiken, die an die Natur angelehnt sind. Typische Phänomene der Natur, die sich in diesen Metaheuristiken wiederfinden, sind die Evolution, die Schwarmintelligenz sowie physikalische Effekte wie Abkühlungseffekte. Nachfolgend werden die auf der Evolution basierenden *evolutionären Algorithmen* sowie die auf der Schwarmintelligenz basierende *Partikelschwarmoptimierung* näher betrachtet. Eine Metaheuristik, die auf physikalischen Effekten basiert, ist das *Simulated Annealing* [Kirkpatrick (1984)].

2.3.2.1 Evolutionäre Algorithmen

Evolutionäre Algorithmen bezeichnen eine Klasse innerhalb der stochastischen, metaheuristischen Optimierungsverfahren. Daher gilt, dass der Algorithmus das Auffinden der optimalen Lösung nicht garantiert, jedoch für eine Vielzahl unterschiedlicher Problemstellungen innerhalb kurzer Zeit ausreichende Ergebnisse erzielt. Verfahren dieser Klasse verwenden die aus der Natur bekannten Prinzipien der Evolutionstheorie nach Darwin. Dazu werden potentielle Lösungskandidaten mit Hilfe geeigneter Operationen künstlich evolviert. Zu den verwendeten Operatoren zählen die *Selektion*, die *Rekombination* sowie die *Mutation* [Hirsch-Kauffmann & Schweiger (1992)]. Die Kostenfunktion wird im Kontext der evolutionären Algorithmen häufig auch Fitnessfunktion genannt. Es ist zu beachten, dass ein niedriger Fitnessfunktionswert eine hohe Fitness bedeutet und somit als „besser“ bewertet wird.

Evolutionäre Algorithmen sind, ebenso wie das in Abschnitt 2.3.1.2 beschriebene Nelder-Mead-Verfahren, ableitungsfreie Verfahren. Innerhalb der Klasse der evolutionären Algorithmen wird zwischen den Verfahren *Genetische Programmierung*, *Genetischer Algorithmus*, *Evolutionsstrategien* sowie *Evolutionäre Programmierung* unterschieden [Nissen (1997)]. Die wesentlichen Unterschiede zwischen den evolutionären Algorithmen liegen in der Wichtigkeit der Operatoren sowie der Repräsentation des Individuums. Während der Genetische Algorithmus die Rekombination als wichtigsten Operator verwendet und die Mutation nur als Hintergrundoperator fungiert, betonen die Evolutionsstrategien die Mutation und verwenden die Rekombination untergeordnet. Insgesamt kommt der Genetische Algorithmus dem Vorbild der Natur am nächsten [Nissen (1997)].

Im Rahmen dieser Arbeit wird der Genetische Algorithmus in Abschnitt 3.2 zur Lösung von nichtlinearen gemischt-ganzzahligen Optimierungsproblemen für die automatisierte Auslegung von Systemen verwendet. Reellwertige Optimierungsvariablen können beispielsweise Reglerparameter sein, während die Ermittlung einer optimalen Systemkomponente (z.B. eines Ventils) auf eine ganzzahlige Optimierungsvariable führt. Der Ablauf des Genetischen Algorithmus ist in Abbildung 2.9 dargestellt. Zu Beginn des Algorithmus wird eine Startpopulation bestehend aus mehreren Individuen zufallsbasiert innerhalb des zulässigen Gebietes erzeugt. Ein Individuum

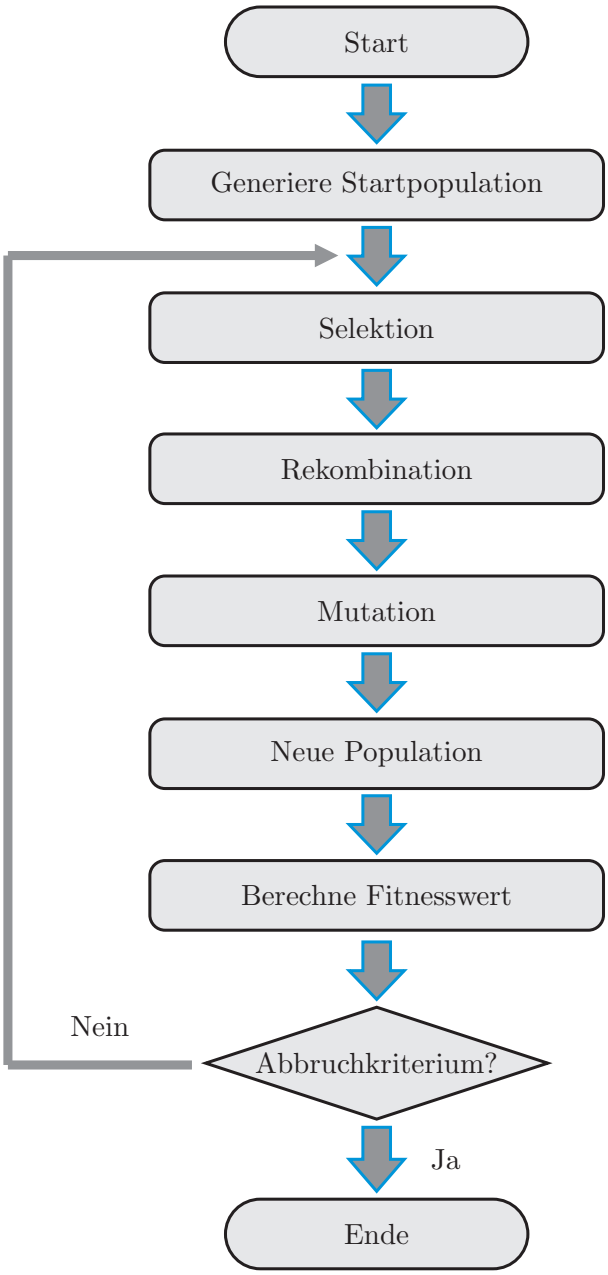

Abbildung 2.9: Ablauf des Genetischen Algorithmus

repräsentiert eine mögliche Lösung des Optimierungsproblems und beinhaltet eine Kombination von Werten für jede Optimierungsvariable. Die Individuen können in unterschiedlicher Form dargestellt werden, etwa als Vektor von reellen Zahlen, aber auch als Bitstrings oder noch allgemein als Objekte jeder Art. Für jedes Individuum wird zunächst der Wert der Fitnessfunktion berechnet. Anhand des Fitnesswertes erfolgt im nächsten Schritt die Selektion, bei der einzelne Individuen mit einem guten Fitnessfunktionswert ausgewählt werden.

Diese selektierten Individuen werden im Schritt der Rekombination dazu verwendet, Nachkommen zu erzeugen. Im einfachsten Fall entsprechen die Werte der einzelnen Optimierungsvariablen des Nachkommen den Mittelwerten der entsprechenden Werte der Eltern. Anschließend werden einzelne Individuen zufallsbasiert innerhalb der Mutationsoperation verändert, bevor eine neue Generation aus sowohl Individuen der alten Generation, den erzeugten Nachkommen und den mutierten Individuen entsteht. Diese Schritte werden so lange wiederholt, bis ein Abbruchkriterium erfüllt ist. Die Wahl des Zeitpunktes, an dem die Iteration am besten abgebrochen wird, ist nicht trivial. Neben dem Erreichen einer vorgegebenen maximalen Anzahl an Generationen wird in [Beasley *et al.* (1993)] vorgeschlagen, das jeweils beste Individuum einer jeden Generation zu speichern. Hat sich das beste Individuum innerhalb der letzten n Generationen nicht verbessert, wird dieses als Lösung verwendet. Die Literatur schlägt für n einen Wert zwischen 5 und 20 vor. Für komplexe Systeme ist ein vergleichsweise hoher Wert zu verwenden, da die Konvergenz für solche Systeme langsamer ist als bei einfachen Systemen.

Für eine detaillierte Beschreibung der einzelnen Operationen und ein Beispiel, an dem die Funktionsweise des Genetischen Algorithmus deutlich wird, sei an dieser Stelle auf [Menager (2012)] verwiesen. Da die Struktur des Individuums nicht vorgeschrieben ist, kann bei Verwendung unterschiedlicher Datentypen für die einzelnen Optimierungsvariablen auf einfach Art und Weise der Algorithmus für die Lösung gemischt-ganzzahliger Probleme angepasst werden.

2.3.2.2 Partikelschwarmoptimierung

Die Partikelschwarmoptimierung (PSO) wurde erstmals im Jahr 1995 von R. Eberhart und J. Kennedy vorgestellt [Eberhart & Kennedy (1995)]. Sie ist ein metaheuristisches, stochastisches Optimierungsverfahren, welches wie die evolutionären Algorithmen auf Naturphänomenen basiert. In der Natur existieren viele Systeme, in denen sich vergleichsweise einfache Lebewesen zu einer Gruppe zusammenschließen und durch gemeinsames Handeln komplexe Probleme lösen. Dieses gemeinsame, koordinierte Handeln wird als *Schwarmintelligenz* bezeichnet [Kramer (2009)]. Die Schwarmintelligenz basiert auf zwei wesentlichen Konzepten, der *Stigmergie* und der *Emergenz*. Stigmergie bezeichnet die Kommunikation der einzelnen Lebewesen untereinander mithilfe der Umwelt, während Emergenz das Phänomen beschreibt, dass einzelne primitive Lebewesen in einem Schwarm ein insgesamt intelligentes Verhalten aufweisen, ohne dass es von außen gesteuert wird. Der Austausch untereinander mit Hilfe der Umwelt wird beispielsweise bei Ameisen beobachtet, die beim Beschreiten eines Pfades Pheromone hinterlassen. Diese Duftstoffe werden von anderen Ameisen wahrgenommen, welche sich abhängig von der wahrgenommenen Pheromonkonzentration schließlich für einen bestimmten Weg entscheiden.

Bei dem PSO-Verfahren wird das Verhalten einer Population von Individuen wie beispielsweise Fischen oder Vögeln simuliert. Die Individuen werden häufig auch

als Partikel bezeichnet. Das Verfahren basiert auf den drei Säulen der *Evaluation*, des *Vergleichs* sowie der *Imitation* [Melchior (2008)]. Die Evaluation stellt die Grundlage einer zielgerichteten Aktion dar, denn ohne eine kontinuierliche Bewertung der einzelnen Individuen ist eine Verbesserung nicht möglich. Die Bewertung geschieht ebenso wie bei den evolutionären Algorithmen durch eine als Fitnessfunktion bezeichnete Kostenfunktion. In der Natur orientieren sich die Individuen stets an ihren Schwarmnachbarn und tauschen Informationen mit diesen aus. Dieser Informationsaustausch zwischen den einzelnen Individuen ist ein wichtiges Unterscheidungskriterium zu den evolutionären Algorithmen, bei denen die Individuen innerhalb einer Generation autark agieren und selbstständig miteinander keinerlei Information austauschen. Innerhalb des PSO-Algorithmus kennt jedes Individuum seinen eigenen Zustand, also seine momentane *Position*, die *Geschwindigkeit*, mit der es sich bewegt, sowie seine *beste gefundene Lösung*. Zusätzlich ist das Individuum in der Lage, in Abhängigkeit der gewählten Nachbarschaftsstrategie mit entsprechenden Schwarmnachbarn die jeweils besten Positionen auszutauschen. Dieser Vergleich des eigenen Zustands mit dem Zustand der anderen Schwarmteilnehmer ist die zweite wichtige Eigenschaft des PSO-Algorithmus. Die Imitation bewirkt schließlich, dass die aus Evaluation und Vergleich gewonnenen Erkenntnisse zielgerichtet eingesetzt werden. Dazu werden erfolgsversprechende Richtungen von Nachbarn imitiert, während wenig erfolgsversprechende Gebiete gemieden werden.

Der Ablauf des Algorithmus ist in Abbildung 2.10 gezeigt. Zu Beginn des Algorithmus wird der Initialschwarm zufallsbasiert innerhalb des zulässigen Suchraumes erzeugt. Die Position eines Individuums X_i beschreibt jeweils eine zulässige Lösung des Optimierungsproblems und beinhaltet Werte für sämtliche Optimierungsvariablen. Neben der Position erhalten die Individuen zufallsbasierte Anfangsgeschwindigkeiten V_i. Für jedes Individuum des Initialschwarms wird vorab der Fitnesswert berechnet. Anschließend wird die Geschwindigkeit für die aktuelle Iteration bestimmt. Dies geschieht mit Hilfe der Formel

$$V_i^t = \omega \cdot V_i^{t-1} + c_1 \cdot \epsilon_1 \cdot (p_i^{t-1} - X_i^{t-1}) + c_2 \cdot \epsilon_2 \cdot (p_g^{t-1} - X_i^{t-1}). \tag{2.39}$$

Wie in der Formel ersichtlich ist, besteht die Geschwindigkeit der aktuellen Iteration aus drei Komponenten. Die erste Komponente ist die Geschwindigkeit der vorherigen Iteration V_i^{t-1}. Der Anteil der alten Geschwindigkeit an der neuen Geschwindigkeit kann mit Hilfe des Gewichtungsfaktors ω vorgegeben werden. Dieser Gewichtungsfaktor wird auch als *Inertia Weight* bezeichnet. Der zweite Anteil ist proportional zur Differenz aus der bisherigen persönlichen besten Position p_i^{t-1} und der vorherigen Position X_i^{t-1}. Für die Gewichtung dieses Summanden steht der Koeffizient c_1 zur Verfügung, ϵ_1 ist eine Zufallszahl mit einem Wert zwischen 0 und 1. Der dritte Summand ist proportional zur Differenz aus der besten Position der Nachbarn p_g^{t-1} und der vorherigen Position X_i^{t-1}. Der Faktor ϵ_2 ist ebenfalls eine Zufallszahl zwischen 0 und 1, der Gewichtungsfaktor des dritten Summanden ist die Konstante c_2. Die beiden Koeffizienten c_1 und c_2 können als Beschleunigungsfakto-

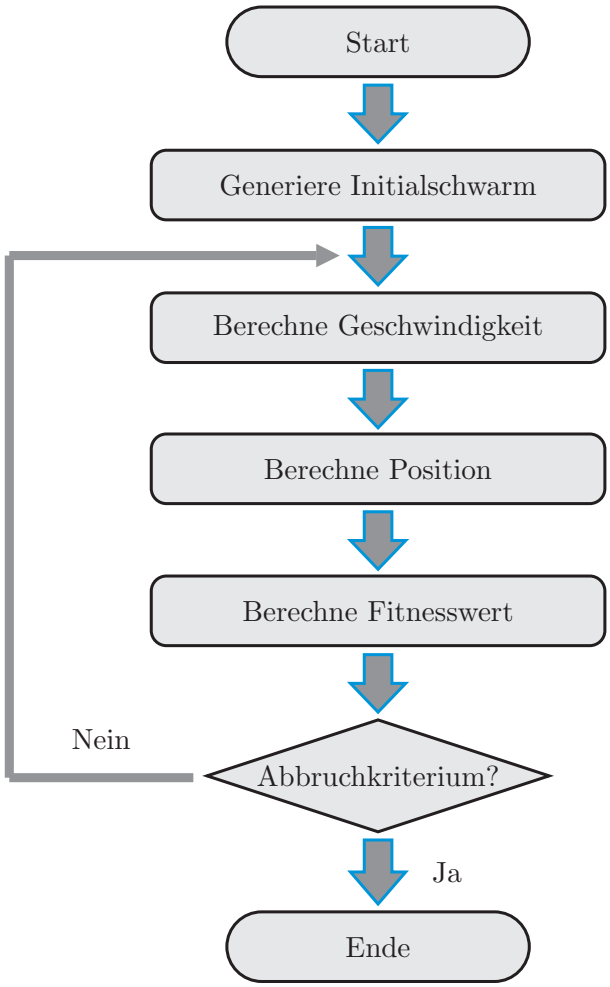

Abbildung 2.10: Ablauf der Partikelschwarmoptimierung

ren in die jeweilige Richtung angesehen werden und gewichten die Entscheidung, ob sich das Individuum eher in Richtung der eigenen besten Lösung oder der globalen besten Lösung bewegt. Er kann daher auch als Einfluss der kognitiven (Faktor c_1) bzw. der sozialen (Faktor c_2) Komponente interpretiert werden [Eitel (2015)]. Der Inertia Weight Faktor ω ist in der Regel keine Konstante. Stattdessen wird zu Beginn der Optimierung ein hoher Wert verwendet, der mit steigender Anzahl an Iterationen verringert wird [Matsui *et al.* (2008)]. Abbildung 2.11 zeigt die Berechnung der neuen Partikelposition graphisch. Abhängig von dem Wert der Zufallszahlen kann sich das Individuum nach der Iteration innerhalb des hellgrün dargestellten Rechtecks aufhalten. Mit Hilfe der alten Position und der neu berechneten Geschwindigkeit, die als Richtungsänderung der aktuellen Periode interpretiert wird, kann die neue Position der aktuellen Iteration bestimmt werden. Dies geschieht mit Hilfe der Gleichung

$$X_i^t = X_i^{t-1} + V_i^t. \tag{2.40}$$

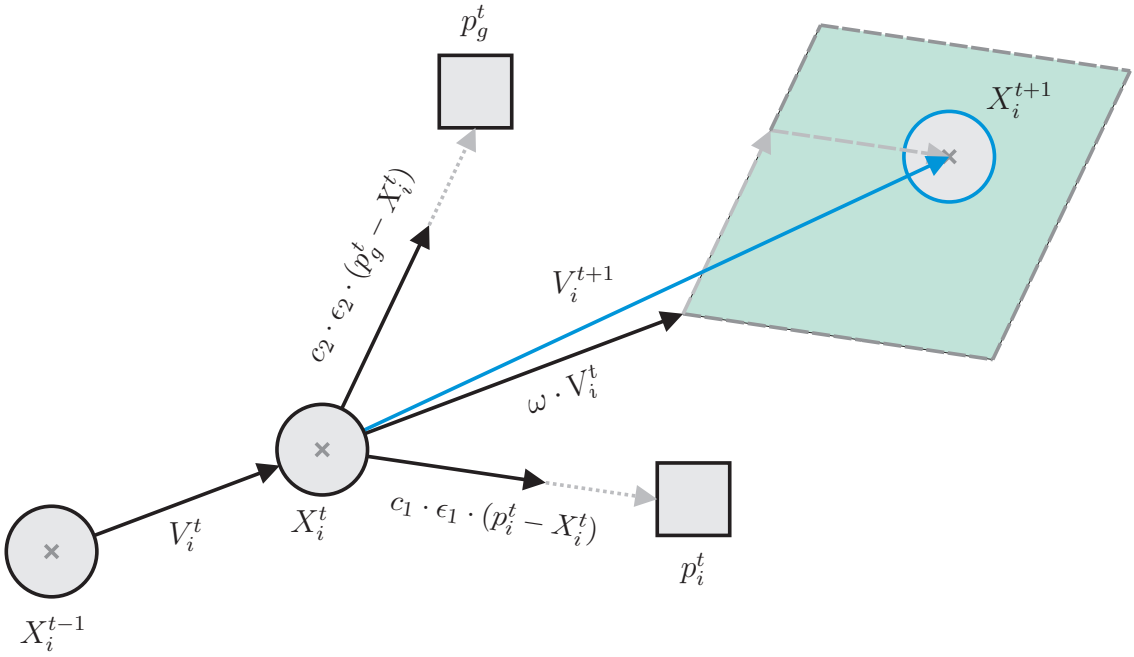

Abbildung 2.11: Berechnung der neuen Partikelposition, frei nach [Eitel (2015)] und [Melchior (2008)]

Für die neue Position wird im letzten Schritt der Fitnesswert bestimmt. Bei Vorliegen einer neuen besten Position wird diese innerhalb der Variablen p_i, die die persönliche beste Position des Individuums beinhaltet, gespeichert. Ist ein Abbruchkriterium erfüllt, endet der Algorithmus und die beste Position des Schwarms ist die berechnete Lösung des Optimierungsproblems. Ansonsten beginnt eine neue Iteration.

In der originalen Implementierung von Eberhart und Kennedy wird eine *Globale Nachbarschaftsstrategie* verwendet. Das bedeutet, dass jedes Individuum mit allen Individuen des Schwarms benachbart ist und mit diesen Daten austauscht. Insbe-

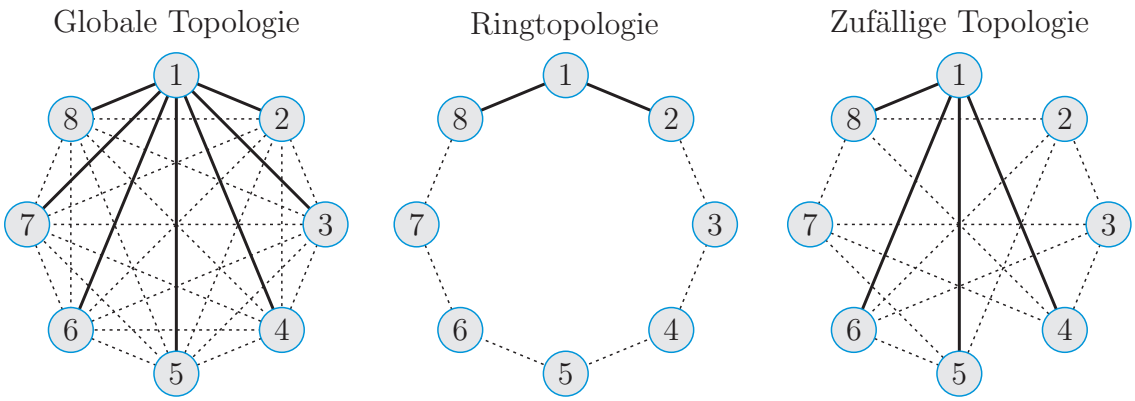

Abbildung 2.12: Verschiedene Nachbarschaftsstrategien innerhalb der Partikelschwarmoptimierung [Eitel (2015)]

sondere kennt jedes Individuum somit die global beste Position des Schwarms, die beste Position des Nachbarn p_g entspricht für jedes Individuum der globalen besten Position des Schwarmes. Dies führt dazu, dass der gesamte Schwarm sich schnell hin zu einem identifizierten Optimum bewegt. Dies birgt die Gefahr, schnell ungewollt in lokale Optima zu gelangen. Neben der Globalen Topologie existieren mittlerweile eine Vielzahl weiterer Strategien, wie die *Ringtopologie* oder die *Zufällige Topologie*. Eine Übersicht über die wichtigsten Nachbarschaftsbeziehungen, die im Rahmen dieser Arbeit verwendet werden, ist in Abbildung 2.12 dargestellt. Zusätzliche Nachbarschaftsbeziehungen sowie weitere Informationen finden sich in [Melchior (2008)].

Der ursprünglich von Eberhart und Kennedy vorgestellte Algorithmus erlaubt lediglich die Verwendung von reellwertigen Optimierungsvariablen. Im Rahmen der automatisierten Auslegung von Systemen treten jedoch häufig gemischt-ganzzahlige Optimierungsprobleme auf, bei denen ein Teil der Variablen reellwertig ist, während der andere Teil diskrete Werte annimmt. Um derartige Probleme lösen zu können, ist der originale Algorithmus zu erweitern.

Erweiterung für gemischt-ganzzahlige Optimierungsprobleme Eine erste Erweiterung wurde 1997, zwei Jahre nach Erscheinen des originalen Algorithmus, ebenfalls von Kennedy und Eberhart vorgestellt [Kennedy & Eberhart (1997)]. Die Erweiterung des ursprünglichen Algorithmus basiert auf der Verwendung von binären Datentypen zur Repräsentation der Optimierungsvariablen. Die zuvor beschriebenen Konzepte der Position sowie der Geschwindigkeit bleiben erhalten, sie erlangen jedoch eine abweichende Bedeutung. Die Geschwindigkeit wird in diesem Kontext als Wahrscheinlichkeit verstanden, dass sich ein Bit innerhalb des Bitstrings verändert. Die Position bezeichnet den Zustand des Bits. Die Geschwindigkeit kann, da sie als Wahrscheinlichkeit interpretiert werden, Werte zwischen 0.0 (0% Wahrscheinlichkeit eines Wechsels des Bits) und 1.0 (100% Wahrscheinlichkeit eines Wechsel des Bits) besitzen. Die Überführung des errechneten Geschwindigkeitswertes in die Wahrscheinlichkeit erfolgt unter Verwendung der Sigmoidfunktion, einer logistischen Verteilungsfunktion. Die neue Position lässt sich auf Basis dieses Konzeptes durch die Anweisung

$$\begin{aligned} &if(rand() < S(V_i^t)) \quad then \quad X_i^t = 1; \\ &else \quad X_i^t = 0; \end{aligned}$$

beschreiben, wobei rand() eine normalverteilte Zufallszahl im Intervall [0.0, 1.0] und $S(V_i)$ die sigmoidale Transformation des berechneten Geschwindigkeitswertes beschreibt.

Eine alternative, sehr einfache Methode zur Verwendung von diskreten Optimierungsvariablen innerhalb der Partikelschwarmoptimierung stellt der Rundungsoperator dar. Auf diese Weise können auch gemischt-ganzzahlige Optimierungsprobleme behandelt werden. Die diskreten Optimierungsvariablen werden zunächst als reell-

wertige Variablen verwendet und ein entsprechender Positionswert ermittelt. Dieser wird anschließend auf einen ganzzahligen Wert gerundet. In [Zhao *et al.* (2013)] wird ein Ansatz vorgestellt, der den Rundungsoperator mit Wahrscheinlichkeiten verknüpft.

Ein weiterer Ansatz findet sich in [Kitayama *et al.* (2006)]. Es wird eine Methode vorgestellt, diskrete Optimierungsvariablen mit Hilfe einer Straffunktion zu behandeln. Diskrete Variablen werden innerhalb des Algorithmus als reellwertige Variablen verwendet, die Kostenfunktion wird jedoch um einen Term erweitert, der den Abstand des Wertes der Variablen zur nächsten Ganzzahl bewertet. Je weiter der Wert einer diskreten Optimierungsvariablen von einer Ganzzahl entfernt ist, desto höher ist der Kostenfunktionswert. Es wird gezeigt, dass der Ansatz für eine Vielzahl von Optimierungsproblemen anwendbar ist, jedoch wird für eine Erhöhung der Anzahl an diskreten Optimierungsvariablen die Kostenfunktion stetig komplexer.

KAPITEL 3

Effiziente Methoden zur Auslegung und Inbetriebnahme

Auf Basis des zuvor beschriebenen Konzeptes der objektorientierten Modellbildung und der behandelten Verfahren zur Lösung von Optimierungsproblemen werden in diesem Kapitel Methoden vorgestellt, die zu einer Reduzierung der Auslegungs- und Inbetriebnahmezeit während der Produktentwicklung führen. Dafür wird zunächst der klassische Entwicklungsprozess für mechatronische Systeme nach VDI Richtlinie 2206 betrachtet und darauf aufbauend ein modellbasierter Entwicklungsprozess vorgestellt. Anschließend werden im Rahmen dieser Arbeit zu entwickelnde Bausteine zur Umsetzung des Entwicklungsprozesses in der Praxis definiert und entwickelt.

3.1 Entwicklungsprozesse für mechatronische Systeme

Heutige Märkte sind geprägt von hohen Innovationsgeschwindigkeiten, kurzen Entwicklungs- und Produktlebenszyklen sowie steigenden Kundenanforderungen bezüglich Leistung, Qualität und Preis der Produkte [VDI2206 (2004)]. Um den Anforderungen gerecht zu werden, ist eine Abkehr von konventionellen Entwicklungsprozessen notwendig, da die Komplexität der gefragten Produkte mit diesen Entwicklungsprozessen nicht länger zu bewältigen ist. Diese sind in der Regel Systeme, die sich durch ein Zusammenwirken von Komponenten mehrerer unterschiedlicher Domänen (z.B. Mechanik, Elektronik, Informationsverarbeitung) auszeichnen. Für die Entwicklung derartiger domänenübergreifender Produkte steht die im Jahr 2004 veröffentlichte VDI Richtlinie 2206 des Vereins Deutscher Ingenieure zur Verfügung [VDI2206 (2004)]. Während sich seit dieser Zeit, gerade im Kontext von Industrie

4.0, die Anforderungen an Produkte weiterhin verändert haben, sind die Entwicklungsprozesse noch immer unverändert.

Die im Rahmen der *Hightech-Strategie 2020* der Deutschen Bundesregierung und der Industrie ausgerufene vierte industrielle Revolution[1] (Industrie 4.0) befasst sich mit der Informatisierung der Fertigung und mündet unter anderem in einer *intelligenten Fabrik* (Smart Factory). Durch die weltweite Vernetzung von Produktionssystemen mit Lagersystemen und Betriebsmitteln gelingt es, dass sich gesamte Fabriken durch den eigenständigen Austausch von intelligenten Maschinen untereinander selbst optimieren. Dadurch lassen sich Produktionsprozesse und Lieferketten effizienter gestalten. Möglich wird dies durch den Einsatz *cyber-physischer Systeme* [Brodtmann & Malorny (2014)]. Cyber-physische Systeme sind gekennzeichnet durch eine Verknüpfung von realen (physischen) Objekten mit informationsverarbeitenden (virtuellen) Objekten über offene, teilweise globale und jederzeit miteinander verbundene Informationsnetze [Bettenhausen & Kowalewski (2013)]. Damit sind cyber-physische Systeme eine Erweiterung der mechatronischen Systeme und zeichnen sich insbesondere dadurch aus, dass diese über eine Dateninfrastruktur wie beispielsweise das Internet oder das Intranet miteinander kommunizieren können. Dadurch ergibt sich abermals eine deutliche Erhöhung der Systemkomplexität. Insbesondere steigt der Anteil an Software innerhalb der Produkte, da diese nicht mehr nur zur Steuerung eines Gerätes oder zur Ausführung eines konkreten Schrittes in das Produkt oder das Produktionssystem eingebettet wird. Stattdessen werden verschiedene Produkte und Dienstleistungen über Software miteinander sowie mit der Umwelt derart verknüpft, dass sich daraus ganz neue Produkte und Dienstleistungen entwickeln lassen [Sendler (2013)]. Es wird in diesem Fall auch von *Smart Services* gesprochen.

Die Entwicklung cyber-physischer Systeme lässt sich aufgrund der hohen Komplexität der Komponenten sowie der Notwendigkeit kurzer Entwicklungszyklen nur mit Hilfe von modellbasierten Entwicklungsmethoden bewältigen. Modellbasierte Entwicklungsmethoden basieren auf einer durchgängigen Begleitung des gesamten Entwicklungsprozesses durch Modelle. Wird das durchgängige modellbasierte Engineering konsequent über sämtliche Phasen des Entwicklungsprozesses (Auslegung, Fertigung, Inbetriebnahme, Betrieb) verfolgt, lassen sich deutliche Effizienzsteigerungen innerhalb der Entwicklung erzielen, die letztendlich zu einer Reduzierung der Time-to-Market führen. Zwei wesentliche Bestandteile der modellbasierten Entwicklungsmethoden sind die *Codegenerierung* zur Vermeidung von Re-Implementierungen und zur durchgängigen Verwendung bereits erlangten Wissens innerhalb der Entwicklung sowie die *virtuelle Inbetriebnahme*, die ein deutlich

[1] Als *erste industrielle Revolution* wird die Erfindung der Dampfmaschine, durch die erstmals händische Arbeit durch mechanische Maschinen ersetzt wurde, bezeichnet (Mechanisierung durch Wasser und Dampfkraft). Als *zweite industrielle Revolution* gilt die Fließbandfertigung, durch die mit Hilfe elektrischer Energie eine deutliche Produktivitätssteigerung in der Produktion erzielt werden konnte. Die *dritte industrielle Revolution* ist durch den Einzug von Computern, Robotik und Automatisierung in die Fertigung gekennzeichnet.

früheres Testen der Softwarekomponenten ermöglicht. Die beiden Methodiken werden nachfolgend genauer betrachtet.

Die zur Umsetzung des modellbasierten Engineerings benötigten Modelle unterscheiden sich sehr stark dahingehend, welche Domäne betrachtet wird. In der Domäne der Mechanik wird beispielsweise häufig auf CAD-Modelle oder FEM-Modelle zurückgegriffen, um Aussagen über Festigkeiten der Bauteile treffen zu können. Zu diesem Zweck stehen eine ganze Reihe spezieller CAD-Tools sowie FEM-Tools zur Verfügung. Andere Domänen verwenden ebenso ihre eigenen Tools und Entwicklungsprozesse. Das Gesamtsystemverhalten ergibt sich schließlich durch die Integration der verschiedenen domänenspezifischen Komponenten. Um das Verhalten dieses Gesamtsystems analysieren zu können, ist die Zusammenführung sämtlicher domänenspezifischer Modelle in einer Umgebung notwendig. Die Verwendung von Modellen über Toolgrenzen hinweg ist im Allgemeinen nur mit Hilfe von Codegenerierung bzw. standardisierten Schnittstellen und Formaten möglich. Aber auch innerhalb einer Domäne ist es wünschenswert, bereits modelliertes Wissen in folgenden Entwicklungsschritten weiterverwenden zu können. Steht beispielsweise bereits ein CAD-Modell einer zu entwickelnden Anlage bereit, so enthält dieses bereits wichtige Informationen des Gesamtsystems wie Trägheitsmomente und Abmaße von einzelnen Komponenten. Bei einem weitergehenden Aufbau eines Mechanikmodells, beispielsweise zur Durchführung einer Systemsimulation, sollten diese Informationen automatisch in das neue Modell übertragen werden, anstatt dieselben Informationen erneut in ein neues Modell zu integrieren. Idealerweise wird ein Modell mit zunehmendem Entwicklungsfortschritt lediglich erweitert. Die dazu notwendige automatische Überführung des Wissens von einem Modell in ein anderes lässt sich erneut nur auf Basis der Methodik der Codegenerierung realisieren. Auf diese Weise ist es beispielsweise möglich, aus einem CAD-Modell automatisch ein Mechanikmodell in der Modellierungssprache Modelica oder ausführbaren Code aus einem in Modelica modellierten Regelalgorithmus zu erzeugen.

Ein zweiter großer Vorteil, der sich durch die Verwendung modellbasierter Entwicklungsmethoden ergibt, ist die Möglichkeit einer virtuellen Inbetriebnahme. Wie bereits in Abschnitt 1.1 beschrieben, macht die Phase der Inbetriebnahme bereits heute bis zu 25% der gesamten Projektdauer aus. Der Hauptanteil daran wird durch das Auffinden und Beseitigen von Softwarefehlern verursacht. Durch den generell stetig steigenden Anteil von Software am gesamten Entwicklungsaufwand (vgl. Abbildung 3.1) wird die Bedeutung der Phase der Inbetriebnahme in Zukunft weiter zunehmen und zu einer Schlüsselphase im Rahmen der Produktentwicklung werden. Bei Verwendung eines klassischen Entwicklungsprozesses ohne virtuelle Inbetriebnahme kann mit der Inbetriebnahme erst begonnen werden, nachdem die Anlage komplett gefertigt, montiert und verkabelt ist. Zu diesem Zeitpunkt steht die Anlage in der Regel bereits beim Kunden und verursacht durch gebundenes Kapital sowie gebundene Räumlichkeiten Kosten. Wird bei der Inbetriebnahme ein Fehler in der Software entdeckt, muss dieser lokalisiert und behoben werden, wodurch sich

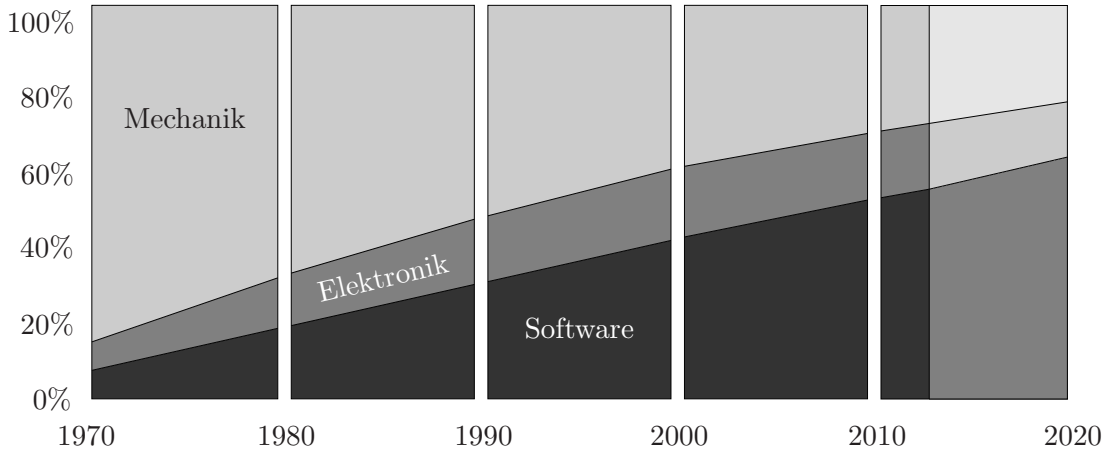

Abbildung 3.1: Anteil der Mechanik, Elektronik und Software am Entwicklungsaufwand; Quelle: VDMA, Zukunftsprognose der ITQ GmbH auf Basis von Marktdaten, nach [Gausemeier (2010)]

die Stillstandszeit der Anlage entsprechend verlängert. Bei Verwendung modellbasierter Entwicklungsmethoden wird während der Auslegung der Anlage jedoch ein Simulationsmodell der Anlage erzeugt. Gelingt es, statt der realen Anlage das virtuelle Abbild für die Inbetriebnahme zu verwenden, kann mit dieser deutlich eher begonnen werden. Die Durchführung einer virtuellen Inbetriebnahme sorgt dadurch für eine Verschmelzung der beiden Phasen der Auslegung und Inbetriebnahme, die zuvor strikt getrennt waren. Das Testen, Erproben sowie Optimieren der zum Betrieb der Anlage benötigten Software, also die Phase der Inbetriebnahme, erfolgt unmittelbar zusammen mit der Auslegung. Die möglicherweise in der Software vorhandenen Fehler können auf diese Weise früher erkannt und beseitigt werden. Neben dieser Zeitersparnis ergibt sich durch die virtuelle Inbetriebnahme ein weiterer wichtiger Vorteil. Erfolgen Auslegung und Inbetriebnahme getrennt voneinander, müssen Konstruktionsfehler bei der Auslegung häufig aufwendig durch Software ausgeglichen werden. Werden Auslegung und Inbetriebnahme gemeinsam durchgeführt, bevor die Anlage gefertigt wird, können konstruktive Änderungen an dem Modell problemlos durchgeführt werden. Auf diese Weise ist gewährleistet, dass ein technisch effizientes und optimiertes System gefertigt wird. Nach Errichtung der Anlage ist schließlich aufgrund möglicher Abweichungen zwischen dem Verhalten des virtuellen und des realen Systems nur noch eine Feinjustierung der Reglerparameter durchzuführen, Stillstandszeiten durch auftretende Fehler in der Software können durch die Methode vermieden werden.

Neben den beiden zuvor genannten wesentlichen Bestandteilen der modellbasierten Entwicklungsmethodik, der Codegenerierung sowie der virtuellen Inbetriebnahme, steht mit der Methode der Optimierung ein weiteres Hilfsmittel bereit, die Effizienz des Entwicklungsprozesses deutlich zu verbessern, um die geforderten kurzen Entwicklungszyklen in der Praxis zu realisieren. Zu Beginn des Entwicklungspro-

zesses, nachdem die Anforderungen an das System festgelegt sind, werden geeignete Komponenten zur Erfüllung dieser Anforderungen ausgewählt. Die Auswahl erfolgt heutzutage in der Regel händisch mittels *trial-and-error* unter Zuhilfenahme von Simulationsmethoden. Unter Verwendung seiner Erfahrungswerte und der Definitionen innerhalb des Pflichtenheftes wählt der Ingenieur händisch entsprechende Komponenten aus, die die festgelegten Anforderungen erfüllen sollen. Das zu entwickelnde Gesamtsystem besteht in der Regel aus der Regelstrecke und einem Regler, der auf das dynamische Verhalten der Regelstrecke einwirkt. Für den Regelalgorithmus wird in der Praxis häufig auf eine bewährte, fixe Reglerstruktur gesetzt, weshalb die Auslegungsaufgabe für den Ingenieur aus der Auswahl einer Kombination von Komponenten für die Regelstrecke sowie geeigneten Reglerparametern besteht. Die Methode des trial-and-error hat einige Nachteile. Mit der zunehmend steigenden Komplexität der Systeme steigt ebenfalls die Anzahl der auszuwählenden Komponenten und Parameter. Die erreichte Güte der Auslegung hängt bei Verwendung der trial-and-error Methode wesentlich von der Expertise des Ingenieurs ab, da unter Umständen nicht sofort ersichtlich ist, wie sich die Veränderung eines einzelnen Parameters auf das Systemverhalten auswirkt. Daraus resultiert ein hoher zeitlicher Aufwand für die Auslegung einer Anlage. Auch nach erfolgter manueller Auslegung durch den Ingenieur steht jedoch nicht fest, ob die gewählte Kombination aus Komponenten und Reglerparametern für die Erzielung der Sollvorgabe tatsächlich optimal ist. Wird das Ziel angestrebt, eine stets gleichbleibende und von der Expertise des Ingenieurs unabhängige Güte der Auslegung zu erreichen, muss die Auslegung automatisiert erfolgen. Dazu lässt sich die Auslegungsaufgabe als Optimierungsaufgabe auffassen, welche sich mit mathematischen Optimierungsmethoden, wie sie in Abschnitt 2.3 behandelt wurden, lösen lässt.

Im folgenden Abschnitt 3.1.1 wird untersucht, auf welche Weise die VDI Richtlinie 2206, die aktuell die Grundlage für die Entwicklung mechatronischer Systeme darstellt, den Entwicklungsprozess derartiger Produkte beschreibt. Anschließend wird in Abschnitt 3.1.2 analysiert, inwieweit die VDI Richtlinie 2206 bzw. modellbasierte Entwicklungsmethoden allgemein derzeit in der Praxis umgesetzt werden. Auf Basis der gesammelten Erkenntnisse wird in Abschnitt 3.1.3 ein modellbasierter Entwicklungsprozess vorgestellt. Für eine Anwendung des Prozesses in der Praxis stehen heute jedoch nicht alle benötigten Methoden bereit. Daher werden abschließend Komponenten definiert, die im Rahmen dieser Arbeit zu entwickeln sind.

3.1.1 Entwicklung nach VDI Richtlinie 2206

Für die Entwicklung domänenübergreifender mechatronischer Systeme steht die VDI Richtlinie 2206 des Vereins Deutscher Ingenieure bereit. Die im Jahr 2004 entwickelte Richtlinie *Entwicklungsmethodik für mechatronische Systeme* erhebt den Anspruch, einen praxisorientierten Leitfaden für die Entwicklung derartiger Produkte bereitzustellen. Um dies zu erreichen, wurden eine Vielzahl von Forschungs- und Praxisarbeiten ausgewertet und die Ergebnisse in der Norm zusammengetragen.

Der Hauptfokus der Richtlinie ist die Bereitstellung von Vorgehensweisen, Methoden und Werkzeugen vor allem für die frühe Phase des Systementwurfs [VDI2206 (2004)].

Neben der VDI Richtlinie 2206 existieren weitere Richtlinien, die sich mit Entwicklungsmethodiken für Produkte und Systeme befassen. Das Ziel der VDI Richtlinie 2206 ist es nicht, die bestehenden Richtlinien abzulösen. Vielmehr ist es das Ziel, die Erkenntnisse der vorhandenen Richtlinien in einer einzelnen Norm zu vereinen. Insbesondere soll die Richtlinie ergänzend zu der VDI Richtlinie 2221 [VDI2221 (1993)] sowie der VDI Richtlinie 2422 [VDI2422 (1994)] positioniert werden.

Die 1993 erstmals veröffentlichte VDI Richtlinie 2221 (*Methodik zum Entwickeln und Konstruieren technischer Systeme und Produkte*) beinhaltet allgemeingültige, branchenunabhängige Grundlagen für ein methodisches Entwickeln von technischen Systemen und Produkten. Der Schwerpunkt dieser Richtlinie liegt auf den Bereichen Mechanik sowie Maschinenbau. Die VDI Richtlinie 2221 ist aufgrund der Tatsache, dass sie sehr gestaltungsorientiert ist, nicht für die Entwicklung komplexer mechatronischer Produkte geeignet [Gausemeier & Möhringer (2003)]. Zielsetzung der VDI Richtlinie 2206 ist es, analog zu der Richtlinie 2221, eine Methodik zur Entwicklung mechatronischer Systeme zu liefern.

Die VDI Richtlinie 2422 (*Entwicklungsmethodik für Geräte mit Steuerung durch Mikroelektronik*) aus dem Jahr 1994 besitzt zwar Ansätze zur parallelen Entwicklung von Systemen, jedoch liegt in dieser Norm der Fokus auf der Mikroelektronik. Das Zusammenspiel der unterschiedlichen Domänen in Form eines übergreifenden Systemkonzeptes wird in der Norm wenig detailliert behandelt. Die Entwicklung der einzelnen Komponenten erfolgt zwar parallel zueinander, jedoch sind die Entwicklungen vollständig getrennt. Mechatronische Systeme basieren auf einer starken Kopplung zwischen den verschiedenen Domänen, sodass ein integrierter Entwicklungsprozess nötig ist, der auf einem hohen Kopplungsgrad der einzelnen domänenspezifischen Entwicklungen basiert.

Zahlreiche Studien belegen, dass es keine Optimalform eines Konstruktionsprozesses gibt, dem der Ingenieur strikt folgen kann [Dörner (1994)]. Das Ziel der VDI Richtlinie 2206 ist es daher, eine flexible Vorgehensweise zu präsentieren. Der zentrale Baustein innerhalb der Richtlinie ist das aus der Softwareentwicklung bekannte V-Modell, welches auf die Anforderungen bei der mechatronischen Produktentwicklung angepasst wurde. Das V-Modell wurde bereits in Abbildung 1.2 dargestellt und besteht im Allgemeinen aus den drei Phasen *Systementwurf*, *domänenspezifischer Entwurf* sowie *Systemintegration*. Ausgangspunkt sind die Anforderungen, die sich aus einem konkreten Entwicklungsauftrag ergeben. Diese Anforderungen können sowohl funktional (Gewünschte Outputmenge einer Produktionsanlage) als auch nicht-funktional (Gesamtpreis der Anlage) sein. Die funktionalen Anforderungen beschreiben das Sollverhalten des Systems, an welchem die Funktion des entwickelten Systems später gemessen wird. Im Systementwurf wird ein domänenübergreifendes

Lösungskonzept erarbeitet, welches die wesentlichen physikalischen und logischen Wirkungsweisen des zu entwickelnden Produktes beschreibt. Auf Basis dieses domänenübergreifenden Lösungskonzeptes wird getrennt voneinander eine detaillierte Entwicklung der einzelnen domänenspezifischen Komponenten durchgeführt. In dieser Phase werden domänenspezifische Werkzeuge und Methoden verwendet, um die Funktionsweise der entsprechenden Komponente sicherzustellen. Die Systemintegration führt schließlich die einzelnen Komponenten zu dem Gesamtsystem zusammen, um dessen Funktionsweise untersuchen zu können. Für diese Eigenschaftsabsicherung des Systems werden sowohl die funktionalen als auch die nicht-funktionalen Anforderungen betrachtet.

In der Regel entsteht das komplexe mechatronische Produkt nicht innerhalb eines Durchlaufes des Makrozyklus. Vielmehr sind dafür in der Regel mehrere Iterationen notwendig, in denen jeweils der Reifegrad des Produktes steigt. In einem ersten Zyklus werden beispielsweise erste Wirkprinzipien und/oder Lösungsvorschläge ausgewählt und grobdimensioniert, welches nach Durchlauf des Makrozyklus ein Labormuster ergibt. Auf Basis des Labormusters werden die Lösungen genauer ausgearbeitet, sodass sich ein detaillierterer Funktionsprototyp ergibt. Abhängig von der konkreten Entwicklungsaufgabe sind weitere Iterationen des Makrozyklus notwendig, um ein serienreifes Produkt zu erhalten.

Der Entwicklungsprozess mechatronischer Produkte ist, wie im V-Modell in Abbildung 1.2 dargestellt, durch Methoden der Modellbildung und Simulation zu unterstützen [VDI2206 (2004)]. Dies erfolgt hauptsächlich im Kontext der Eigenschaftsabsicherung, bei der die Funktionsweise der entwickelten Lösung mit der gewünschten Funktionsweise aus der Definition der Anforderungen verglichen wird. Die Verwendung virtueller Prototypen erlaubt es, auf den Bau realer Prototypen zu verzichten und auf diese Weise Zeit und Kosten zu sparen. Ist es nicht möglich, den gesamten Prototypen virtuell abzubilden, beispielsweise weil die Modellierung bestimmter physikalischer Effekte mit äußerst hohem Aufwand verbunden ist, kann eine Kombination aus einem virtuellen und einem realen Prototyp verwendet werden. Die Integration von realen Komponenten und Systemmodellen in eine gemeinsame Simulationsumgebung wird als *Hardware-In-The-Loop* bezeichnet.

Während des Systementwurfs sowie des domänenspezifischen Entwurfs werden eine Reihe unterschiedlicher Modelle benötigt, die jeweils spezielle Effekte abbilden. Hierbei ist zu beachten, dass sowohl innerhalb einer Domäne mehrere Modelle benötigt werden (in der Mechanik z.B. ein CAD-Modell sowie ein Mechanikmodell für die Systemsimulation) als auch Modelle verschiedener Domänen verwendet werden. Innerhalb der unterschiedlichen Disziplinen sind dafür, wie bereits zuvor beschrieben, verschiedene Darstellungsformen gebräuchlich. Das Gesamtsystemverhalten ergibt sich bei mechatronischen Systemen aus der Zusammenführung der verschiedenen Komponenten im Rahmen der Systemintegration. Dies setzt jedoch spezielle Methoden der Modellintegration voraus. Eine in der Richtlinie vorgeschlagene mögliche

Umsetzung der Modellintegration basiert auf der Verwendung mathematischer Beschreibungen für die Modelle. Für eine Umsetzung in der Praxis ist dieser Vorschlag nicht ausreichend, da wesentliche Probleme wie z.B. die Definition von Eingängen und Ausgängen der verschiedenen Modelle sowie die Herstellung der benötigten Kausalität zum Lösen der Gleichungen nicht geklärt sind.

Der Aufwand der Erstellung der Modelle kann reduziert werden, indem die Modelle nicht in jedem Entwicklungsschritt erneut aufgebaut werden, sondern lediglich ergänzt werden. Diese Durchgängigkeit der Modellverwendung über das gesamte V-Modell ermöglicht es, auf vorherigem Wissen aufzubauen und Re-Implementierungen zu vermeiden. Zumindest über die verschiedenen Phasen der Entwicklung hinweg wird diese Methodik der modellbasierten Entwicklung in der VDI Richtlinie 2206 empfohlen. Das V-Modell behandelt jedoch lediglich die frühen Phasen der Auslegung und Entwicklung eines Produktes bzw. Systems, die darüber hinausgehenden Phasen der Inbetriebnahme sowie des Betriebes sind nicht Bestandteil der Richtlinie.

Bewertung der Richtlinie in Bezug auf einen Einsatz in der Praxis Die VDI Richtlinie 2206 stellt einen Entwicklungsprozess speziell für mechatronische Produkte zur Verfügung. Gerade im Kontext von Industrie 4.0 werden in Zukunft jedoch cyber-physische Systeme zunehmend in den Fokus geraten. Um diese Produkte effizient entwickeln zu können, ist ein durchgängiges Engineering notwendig, welches noch deutlich über die in der Richtlinie beschriebene Durchgängigkeit hinaus geht. Die VDI Richtlinie 2206 legt ihren Fokus hauptsächlich auf die frühen Phasen der Produktentwicklung und versucht, diese durch Modelle zu begleiten. Dies betrifft die im V-Modell dargestellten Phasen des Systementwurfs, des domänenspezifischen Entwurfs sowie der Systemintegration. Eine deutliche Effizienzsteigerung ist jedoch zu erzielen, wenn die Durchgängigkeit während der gesamten Entwicklung bis hinein in den Betrieb umgesetzt wird. Bereits die Anforderungen zu Beginn eines Entwicklungsprozesses sollten in Form eines Modells hinterlegt werden. Dazu kann beispielsweise die Modellbeschreibungssprache SysML[1] verwendet werden. Die Generierung weiterführender Modelle aus der abstrakten Beschreibung in SysML ist möglich, eine Erzeugung von Modelica-Modellen ist in [Ji *et al.* (2012)] beschrieben. Die Möglichkeit der Generierung von Modelica-Code aus der Sprache ModelicaML ist beispielsweise in [Schamai *et al.* (2009)] sowie [Kara (2015)] präsentiert. Auf diese Weise wird bereits zu Beginn des Entwicklungsprozesses auf mehrfache Implementierungen von vorhandenem Wissen verzichtet, indem unter anderem Abmaße einzelner Komponenten unmittelbar aus dem Anforderungsmodell in ein nachfolgendes Modell überführt werden. Ein weiteres Beispiel, wie ein Modell des Systementwurfs in ein Modell des domänenspezifischen Entwurfs auf Basis des offenen Standards

[1] Die Systems Modeling Language (SysML) ist eine von der Object Management Group (OMG) entwickelte graphische, auf UML 2.0 basierende standardisierte Modellierungssprache, um Systeme auf einer abstrakten und einheitlichen Weise zu beschreiben.

UML transformiert werden kann, ist in [Gehrke *et al.* (2007)] beschrieben. Des Weiteren existieren Ansätze, wie funktionale Anforderungen auch unmittelbar in ein Modelica-Modell integriert werden können, sodass diese während der Simulation automatisiert geprüft werden [Otter *et al.* (2015)] [Buffoni & Fritzson (2014)]. Die Verwendung von Modellen sollte zusätzlich auch über die Phase der Auslegung hinaus bis in die Inbetriebnahme sowie den Betrieb erfolgen, sodass beispielsweise das Modell eines Regelalgorithmus unmittelbar als Basis für den Regler auf der Steuerungshardware verwendet wird.

Um eine Überführung von modelliertem Wissen über Sprach- bzw. Toolgrenzen hinweg zu gewährleisten, ist es zwingend notwendig, die Modellbeschreibungen auf Basis *offener Standards* zu hinterlegen. Die Verwendung offener Standards ist ein unverzichtbares Konzept, um ein durchgängiges modellbasiertes Engineering in der Praxis umzusetzen. Heutzutage wird für die Erstellung von Simulationsmodellen zumeist auf kommerzielle Tools zurückgegriffen. Diese Werkzeuge sind in der Regel abgeschlossen, sodass die Modelle, die innerhalb der Umgebung erstellt wurden, lediglich innerhalb dieser Umgebung verwendet werden können. Aus diesem Grund wurde im Jahr 2010 das *Functional Mockup Interface* (FMI) vorgestellt. FMI ist eine offene, standardisierte Schnittstelle für den Austausch von Simulationsmodellen zwischen Simulationsumgebungen und wird von der Modelica Association entwickelt. Dieser Standard wird mittlerweile von vielen Umgebungen unterstützt, wobei einige lediglich einen Import oder einen Export von *Functional Mockup Units* (FMUs) ermöglichen[1]. Der FMI Standard ermöglicht es, Modelle aus verschiedenen domänenspezifischen Simulationsumgebungen in eine gemeinsame Umgebung zu überführen. Die exportierte FMU kann entweder nur das Modell oder aber das Modell sowie ein numerisches Lösungsverfahren beinhalten, sodass eine Co-Simulation durchgeführt werden kann [Blochwitz *et al.* (2011)].

Neben dem FMI Standard existieren eine Reihe weiterer Standards für den Datenaustausch zwischen unterschiedlichen Berechnungs- und Simulationstools. Im Jahr 2002 wurde, hauptsächlich aus der Automobilindustrie, der von kommerziellen CAx Standards unabhängige *MechaSTEP* Standard geschaffen. Dieser Standard wurde speziell für den Datenaustausch bei der Entwicklung von mechatronischen Systemen entwickelt und unterstützt die Domänen Elektrik, Hydraulik, Pneumatik, Automation und Mechanik. In der Praxis hat sich dieser Standard jedoch nicht durchgesetzt, obwohl die Eignung beispielsweise speziell für die Domäne der Mehrkörpermechanik nachgewiesen ist [Bellalouna (2009)].

Die Verwendung offener Standards ist nicht nur aus Sicht der Wiederverwendung von Modellen sinnvoll. Besonders kleine und mittelständische Unternehmen haben heutzutage keine Möglichkeit, modellbasierte Entwicklungsmethoden zu nutzen. Die wenigen verfügbaren Tools, die eine durchgängige Entwicklung zumindest in Teilen

[1] Eine Übersicht über die Unterstützung des FMI Standards in gängigen Simulationsumgebungen findet sich bei `https://www.fmi-standard.org/tools`

erlauben, stehen nur gegen hohe Lizenzgebühren zur Verfügung. Ein Beispiel dafür ist die bereits in Abschnitt 1.2 vorgestellte *3DEXPERIENCE Plattform* der Firma Dassault Systèmes. Dieses Werkzeug unterstützt den Aufbau von CAD-Modellen in der ebenfalls von Dassault entwickelten CAD-Umgebung CATIA[1] sowie eine automatische Erzeugung eines Modelica-Modells aus der CAD-Beschreibung. Ebenso sind bereits Anforderungen in diesem Werkzeug hinterlegbar. Die Durchgängigkeit endet jedoch nach Abschluss der Auslegungsphase, eine Überführung von Modellen in die Phase der Inbetriebnahme sowie den Betrieb ist nicht möglich. Weiterhin ist die Verwendung der CAD-Umgebung CATIA zwingend vorgeschrieben, andere CAD-Tools können nicht verwendet werden. Ebenso wie bei der Verwendung der 3DEXPERIENCE Plattform fallen auch für die Benutzung weit verbreiteter Softwarelösungen wie MATLAB/Simulink hohe Lizenzkosten an.

Sowohl für die domänenspezifische Modellbildung als auch die domänenübergreifende Modellierung des Gesamtsystems ist die Verwendung der in Abschnitt 2.1.2 vorgestellten offenen, standardisierten Modellierungssprache Modelica geeignet. Für die Verwendung von Modelica-Modellen existieren neben kommerziellen Umgebungen ebenfalls Open Source Simulationstools, sodass auch kleine und mittelständische Unternehmen in der Lage sind, diese entsprechenden Werkzeuge zu nutzen. Da es sich bei Modelica um eine Sprache und nicht um ein spezielles Tool handelt, besteht bei Verwendung von Modelica keine Abhängigkeit gegenüber speziellen Simulationsumgebungen.

Innerhalb der VDI Richtlinie 2206 ist beschrieben, dass der Entwicklungsprozess mechatronischer Produkte durch die Methoden der Modellbildung und Simulation zu unterstützen ist. Die Auslegung der domänenspezifischen Teilsysteme erfolgt in diesem Fall händisch mittels trial-and-error. Gerade für komplexe Systeme, wie sie im Kontext von Industrie 4.0 zunehmend auftreten, lässt sich die für die Auslegung benötigte Zeit durch die Verwendung einer automatisierten Auslegung auf Basis mathematischer Optimierungsmethoden signifikant reduzieren. Zur konsequenten Umsetzung eines durchgängigen, modellbasierten Engineerings, wie es für die Entwicklung derartiger Produkte notwendig ist, ist der Einsatz von Optimierungsverfahren in der Zukunft unverzichtbar. Dieser Aspekt fehlt in der bisherigen Beschreibung des Produktentwicklungsprozesses innerhalb der VDI Richtlinie 2206 vollständig.

3.1.2 Einsatz modellbasierter Entwicklungsmethoden in der Praxis

Trotz der Bereitstellung einer Richtlinie speziell für die Entwicklung komplexer mechatronischer Produkte und Systeme bestehen in der heutigen Praxis noch erhebliche Defizite bei der Entwicklung derartiger Produkte. Dies liegt in erster Linie daran, dass die in der Richtlinie beschriebenen Ansätze in der Theorie zwar er-

[1] **C**omputer **A**ided **T**hree-Dimensional **I**nteractive **A**pplication

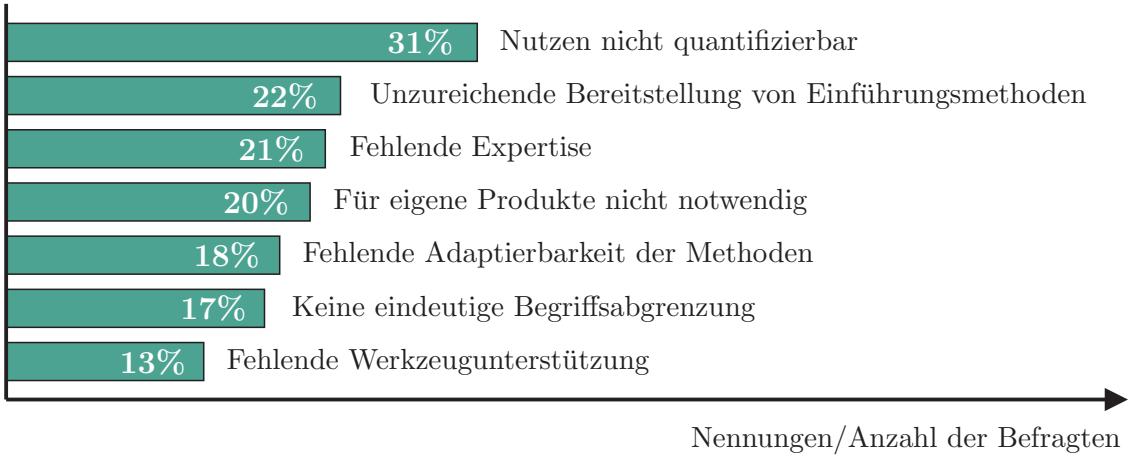

Abbildung 3.2: Hindernisse in der Anwendung des Systems Engineering [Gausemeier *et al.* (2013)]

folgsversprechend sind, bisher aber kaum Möglichkeiten bestehen, diese in der Praxis umzusetzen. Möglichkeiten der Umsetzung werden auch in der VDI Richtlinie 2206 nicht erwähnt. Diese Probleme kommen ebenfalls in zahlreichen durchgeführten Studien zum Vorschein. Eine im Jahr 2006 durchgeführte Studie der Aberdeen Group untersucht die Umsetzung mechatronischer Entwicklungsprozesse in Industrieunternehmen. Diese kommt zu dem Ergebnis, dass 50% der Unternehmen die domänenspezifischen Komponenten mechatronischer Produkte vollständig getrennt voneinander mit jeweils domänenspezifischen Werkzeugen und Prozessen entwickeln. Weitere 30% der Unternehmen verwenden separate Entwicklungsprozesse, die jedoch zumindest einen teilweisen Austausch ermöglichen. Lediglich 20% der befragten Unternehmen verwenden einen gemeinsamen, durchgängigen Entwicklungsprozess [Jackson (2006)].

Dieses Bild hat sich auch sieben Jahre danach nicht signifikant verändert. In einer 2013 veröffentlichten Studie des Heinz Nixdorf Institutes der Universität Paderborn sowie des Fraunhofer-Instituts für Produktionstechnologie IPT wird erneut untersucht, inwieweit die Methoden der modellbasierten Entwicklung innerhalb von Unternehmen angewendet werden. Neun Jahre nach Erscheinen der VDI Richtlinie 2206 ist der Einsatz modellbasierter Entwicklungsmethoden noch immer kein Standard in der Automatisierungsindustrie. Die genannten Gründe, warum die Methodik nicht verwendet wird, sind in Abbildung 3.2 dargestellt. Ein Hindernis ist es, dass der erzielbare Nutzen bei Einsatz modellbasierter Entwicklungsmethoden für die Unternehmen nicht quantifizierbar ist. Dies liegt daran, dass den Entwicklern der Zugang zu den Methoden fehlt und diese in dem Bereich nicht ausreichend Expertise besitzen. Ein wesentliches Ergebnis der durchgeführten Studie (vgl. Abbildung 1.5) ist, dass der größte Handlungsbedarf bei der Bereitstellung durchgängiger Werkzeugketten besteht, mit denen die Entwickler die modellbasierten Entwicklungsmethoden

auf einfache Weise umsetzen können. Das Ziel muss es also sein, die Methoden der modellbasierten Entwicklung den Entwicklern zugänglich zu machen und konkrete Umsetzungsvorschläge zu liefern, die in der VDI Richtlinie 2206 vollständig fehlen. Dies ist gerade im Hinblick darauf, dass es sich bei der Norm um einen praxisorientierten Leitfaden handelt, notwendig. Ein Hauptziel dieser Arbeit besteht darin, einen modellbasierten Entwicklungsprozess zu definieren und fehlende Bausteine zu entwickeln, die eine einfache Umsetzung moderner Entwicklungsmethoden in der Praxis ermöglichen.

3.1.3 Umsetzung eines effizienten modellbasierten Engineerings

Wie in den vorherigen Abschnitten ausführlich beschrieben wurde, bietet der Einsatz modellbasierter Entwicklungsmethoden ein deutliches Potential zur Steigerung der Effizienz innerhalb des Entwicklungsprozesses. Bisher stehen jedoch nur sehr wenige Möglichkeiten bereit, ein solches durchgängiges Engineering in der Praxis umzusetzen, was auch bereits in zahlreichen Studien nachgewiesen werden konnte. Die heute bereitstehenden Möglichkeiten setzen auf kommerzielle Tools und sind daher einer Vielzahl an kleinen und mittleren Unternehmen nicht zugänglich. Ferner ist auch mit diesen Tools die Durchführung eines vollständig durchgängigen Engineerings nicht möglich. Die vorliegende Arbeit stellt daher einen Beitrag dar, wie derartige Engineeringmethoden für die Entwicklung von Produkten in der Praxis eingesetzt werden können.

Der sich aus dieser Arbeit ergebende durchgängige, modellbasierte Engineeringprozess ist in Abbildung 3.3 dargestellt. Der erste Schritt besteht in der Definition der Anforderungen. Anschließend wird ein funktionales Modell abgeleitet, welches die Topologie des Gesamtsystems beschreibt. Nachdem diese feststeht, folgt die Erstellung eines CAD-Modells. Für eine Analyse des dynamischen Verhaltens des Gesamtsystems wird bei einer Anwendung modellbasierter Entwicklungsmethoden parallel zu der Erstellung des CAD-Modells ein physikalisches Gesamtsystemmodell (Regelstrecke sowie Regler) erzeugt. An dieser Stelle kann im Sinne einer Durchgängigkeit in der Entwicklung der mechanische Teil des physikalischen Modells automatisch aus dem CAD-Modell erzeugt werden, da die benötigten Informationen bei einer sorgfältigen Erstellung des CAD-Modells bereits vollständig vorhanden sind. Nachdem das System ausgelegt ist, wird mit der Erstellung der Steuerungsapplikation begonnen. An dieser Stelle ist es wünschenswert, den bereits innerhalb des physikalischen Modells vorhandenen Regelalgorithmus weiterzuverwenden. Dies kann durch eine Generierung von Code aus dem Simulationsmodell erreicht werden, welcher anschließend auf der Steuerungshardware ausgeführt wird. Zur frühzeitigen Validierung der Steuerungsapplikation kann unter Zuhilfenahme des Simulationsmodells der Regelstrecke eine virtuelle Inbetriebnahme durchgeführt werden. Parallel zu der Erstellung der Steuerungsapplikation sowie der virtuellen Inbetriebnahme wird die Anlage gefertigt. In der Phase der Inbetriebnahme wird die Steuerungsapplikation schließlich mit der realen Anlage verbunden und eine finale Anpassung der

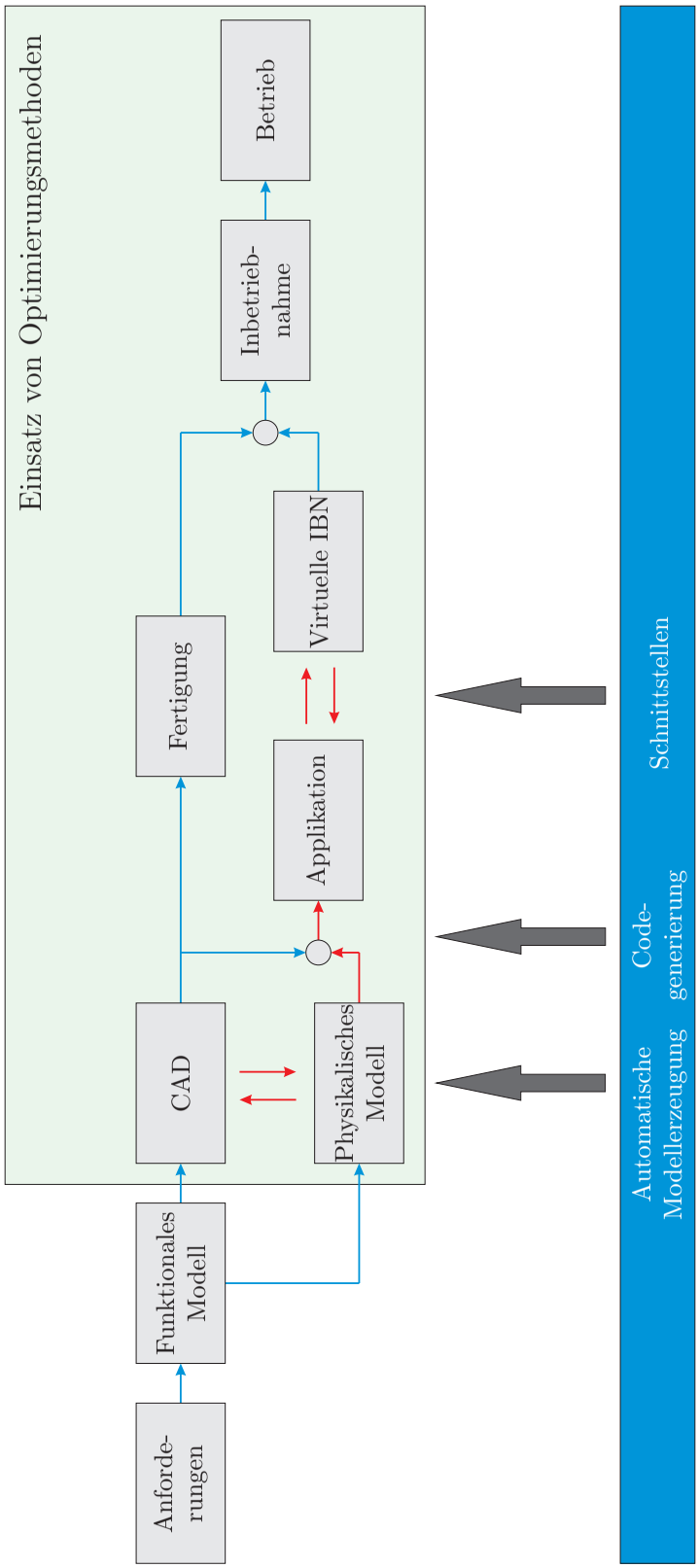

Abbildung 3.3: Modellbasierter Entwicklungsprozess

Steuerungsparameter durchgeführt, eh die Phase des Betriebs erreicht wird. Der Engineeringprozess wird zusätzlich durch Optimierungsmethoden unterstützt, welche für unterschiedliche Zwecke eingesetzt werden können. Neben der Möglichkeit der automatischen Auslegung des Systems, bei der optimale Komponenten zur Erfüllung der Systemanforderungen ermittelt werden, kann auch eine HiL-Optimierung durchgeführt werden, bei der optimale Steuerungsparameter während einer HiL-Simulation unmittelbar auf der Steuerungshardware bestimmt werden. Der Einsatz von Optimierungsmethoden kann ebenfalls während des Betriebs erfolgen, beispielsweise, um die Reglerparameter kontinuierlich zu optimieren, um auch bei Verschleiß in der realen Anlage stets einen optimalen Prozess zu ermöglichen.

Definition der benötigten Komponenten zur Umsetzung Für die Umsetzung des Engineeringprozesses in der Praxis sind im Wesentlichen fünf Komponenten notwendig. Zur Unterstützung in den verschiedenen Prozessphasen wird ein *Optimierungstool* benötigt, welches eine automatisierte Ermittlung der optimalen Kombination aus zu verwendenden Komponenten sowie optimalen Reglerparametern ermöglicht. Die Optimierung geschieht unter Verwendung eines Simulationsmodells des Systems und kann sowohl ausschließlich in der Simulationsumgebung sowie unter Einbeziehung realer Steuerungshardware erfolgen. Die Modellierung der benötigten Modelle erfolgt in der standardisierten, offenen Modellierungssprache Modelica. Für eine effiziente Erstellung des physikalischen Modells des Gesamtsystems wird eine *automatische Modellerzeugung* des dynamischen Mechanikmodells aus einer vorhandenen CAD-Darstellung benötigt. Für die Überführung der Simulationsmodelle aus der Phase der Auslegung in die Phase der Inbetriebnahme ist eine *Werkzeugkette* notwendig, die aus dem Modell ausführbaren Code generiert, der auf der Zielhardware ausgeführt werden kann. Die Entwicklung dieser Werkzeuge muss ebenfalls auf offenen Standards basieren, um eine Abhängigkeit von kommerziellen Tools zu vermeiden und das Werkzeug auch kleineren und mittleren Unternehmen verfügbar zu machen. Für die Ausführung des aus dem Simulationsmodell generierten Codes ist ein *Simulationskern* notwendig, der die numerischen Verfahren beinhaltet. In diesem Kontext sind spezielle numerische Verfahren notwendig, da die Ausführung des Codes auf der Steuerungshardware in Echtzeit ablaufen muss. Abschließend werden *Schnittstellen* benötigt, die eine Einbindung der realen Steuerungshardware in die Simulationsumgebung ermöglichen, um eine virtuelle Inbetriebnahme durchzuführen. Von den genannten fünf Komponenten werden im Rahmen dieser Arbeit vier umgesetzt, die automatische Modellerzeugung aus einer CAD-Repräsentation ist nicht Bestandteil dieser Arbeit. Die übrigen vier Komponenten werden in den folgenden Abschnitten ausführlich behandelt. Im nachfolgenden Abschnitt 3.2 wird die Entwicklung des Optimierungstools dargestellt. Anschließend wird in Abschnitt 3.3 eine Möglichkeit präsentiert, ein Modelica-Modell auf einer Industriesteuerung auszuführen. Bei diesem Modell kann es sich sowohl um ein Reglermodell, um die Methodik des Rapid Control Prototypings umzusetzen, oder ein Modell der Regelstrecke handeln. Im letztgenannten Fall kann beispielsweise eine modellbasierte Dia-

gnose oder eine modellbasierte Vorsteuerung umgesetzt werden. In diesem Abschnitt wird ebenfalls der Simulationskern behandelt und um notwendige Module zur Umsetzung einer Echtzeitsimulation erweitert. Anschließend werden in Abschnitt 3.4 benötigte Schnittstellen dargestellt, um reale Industriehardware in Simulationsumgebungen zu integrieren. In diesem Kontext wird ebenfalls auf die Komplikationen bei der Wahrung der Synchronität zwischen Steuerungshardware und Simulation eingegangen.

3.2 Einsatz von Optimierungsmethoden in der Auslegungsphase

Die Entwicklung eines neuen Produktes beginnt zunächst mit der Definition der Anforderungen. Diese werden in der Regel in Form eines *Lastenheftes* vom Kunden festgehalten. Die innerhalb des Lastenheftes vom Kunden gewünschten Anforderungen werden auf technische Realisierbarkeit geprüft. In Absprache mit dem Kunden entsteht ein *Pflichtenheft*, welches den Entwickler verpflichtet, die darin enthaltenen funktionalen sowie nicht-funktionalen Anforderungen zu realisieren. Das Pflichtenheft enthält somit unter anderem das Sollverhalten des zu entwickelnden Produktes. Der nächste Schritt ist die Konzeptphase, innerhalb der mögliche Konzepte zur Umsetzung der Anforderungen ermittelt werden. Ist ein geeignetes Konzept gefunden, kann in der Auslegungsphase der Entwurf des Produktes beginnen. Der erste Schritt besteht zumeist darin, geeignete Komponenten zu identifizieren, die in ihrer Gesamtheit der Funktionserfüllung dienen sollen. Im Rahmen dieser Arbeit werden vorwiegend technische Anlagen aus dem Industriebereich betrachtet.

In der Auslegungsphase wird bereits heute relativ häufig ein Simulationsmodell des zu entwickelnden Systems erstellt, um frühzeitig die Eigenschaften und Funktionen abzusichern. Je später im Entwicklungszyklus eine fehlerhafte Auslegung erkannt wird, desto höhere Kosten verursacht sie. Die Verwendung von virtuellen Prototypen hat wesentliche Vorteile im Vergleich zum konventionellen Prototypenbau. Neben deutlichen Zeit- und Kostenvorteilen gegenüber der Erstellung realer Prototypen bietet das *Virtual Prototyping* einige weitere Vorteile. Zum einen können Simulationsmodelle bei konstruktiven Änderungen des Systems sehr schnell und einfach angepasst werden. Eine zeitaufwendige Fertigung eines neuen realen Prototypen entfällt. Zum anderen können innerhalb einer Simulationsumgebung Szenarien und Betriebszustände getestet werden, für die bei einer realen Durchführung eine Gefahr für Mensch und Maschine besteht. Ein weiterer Vorteil sind stets gleichbleibende Randbedingungen innerhalb der Simulation, welche bei realen Versuchen häufig schwierig zu erreichen sind. Sollen jedoch Untersuchungen bei unterschiedlichen Umweltbedingungen durchgeführt werden, ist es lediglich notwendig, den entsprechenden Parameter innerhalb des Simulationsmodells zu ändern. Gerade bei räumlich verteilt arbeitenden Teams haben virtuelle Prototypen weiterhin den Vorteil, dass diese an mehreren Standorten gleichzeitig verwendet werden können.

Für die Auswahl geeigneter Komponenten sowie Reglerparametern wird zumeist das Vorgehen mittels trial-and-error eingesetzt, bei dem der Ingenieur die Komponenten und Parameter händisch solange variiert, bis das tatsächliche Systemverhalten dem gewünschten, vorgegebenen Verhalten bestmöglich entspricht. Diese Methode hat jedoch, wie bereits in Abschnitt 3.1 beschrieben, erhebliche Nachteile. Eine stets gleichbleibende und von der Expertise des Ingenieurs unabhängige Güte der Auslegung lässt sich nur durch eine Automatisierung des Auslegungsprozesses erreichen. Dazu lässt sich die Auslegungsaufgabe als mathematisches Optimierungsproblem auffassen. Gesucht sind diejenigen Komponenten und Parameter, die eine Norm der Abweichung zwischen Systemverhalten und vorgegebenem Sollverhalten minimieren. Mathematisch kann das Problem wie folgt beschrieben werden:

$$\min_{\mathbf{x},\mathbf{y}} \sum_{i=1}^{N} \|y_i^{sim}(t,\mathbf{x},\mathbf{y}) - y_i^{soll}(t)\|. \tag{3.1}$$

Dabei beschreibt y_i^{sim} das simulativ ermittelte Systemverhalten, welches neben der Zeit t von den freien Optimierungsvariablen $\mathbf{x}$ und $\mathbf{y}$ abhängig ist, während y_i^{soll} das vorgegebene, gewünschte Verhalten im Zeitverlauf beschreibt. Die Werte für y_i^{soll} sowie y_i^{ist} werden dazu an N unterschiedlichen Zeitpunkten betrachtet. Im Rahmen dieser Arbeit wird für die Norm häufig die *euklidische Norm* verwendet. Bei dem in Gleichung 3.1 dargestellten Optimierungsproblem handelt es sich um ein gemischt-ganzzahliges Optimierungsproblem, d.h. ein Optimierungsproblem, für das ein Teil der Optimierungsvariablen reellwertig ist ($\mathbf{x} \in \mathbb{R}^m$), während der andere Teil ganzzahlig ist ($\mathbf{y} \in \mathbb{Z}^n$). Es ist zu beachten, dass es sich bei der Verwendung von Komponenten als Optimierungsvariablen um diskrete Variablen handelt.

Die in Gleichung 3.1 beschriebene mathematische Beschreibung stellt lediglich eine Möglichkeit eines Optimierungsproblems dar, welches im Rahmen der automatisierten Auslegung eines Systems auftreten kann. Diese Formulierung ist geeignet, wenn das Optimierungsziel darin besteht, dass das tatsächliche Systemverhalten bestmöglich mit einem vorgegebenen Sollverhalten übereinstimmt. Besteht das Optimierungsziel stattdessen darin, auftretende Schwingungen im System zu minimieren, ist die Kostenfunktion entsprechend zu formulieren. Die Bewertung kann mit Hilfe einer *Fourier-Transformation* des entsprechenden Signals erfolgen [Leute (2004)]. Die Fourier-Transformation erlaubt es, beliebige kontinuierliche, aperiodische Signale in ein kontinuierliches Frequenzspektrum zu zerlegen. Das Optimierungsproblem zur Minimierung der Schwingungen eines Signals kann schließlich mathematisch durch

$$\min_{\mathbf{x},\mathbf{y}} \sum_{i=1}^{N} \omega_i \, \hat{y}_i(\mathbf{x},\mathbf{y}) \tag{3.2}$$

beschrieben werden. Dabei beschreibt $\hat{y}_i$ die Amplitude des Signals bei der Frequenz i. Das Signal selbst ist dabei natürlich abhängig von den gewählten Werten

für die Optimierungsparameter. Die Amplituden des Signals für die verschiedenen betrachteten Frequenzen werden zur Bestimmung des Kostenfunktionswertes aufsummiert. Unter Umständen ist es gewünscht, die Amplituden mit Hilfe des Faktors ω entsprechend zu gewichten, sofern bestimmte Frequenzbereiche innerhalb der Kostenfunktion stärker berücksichtigt werden sollen.

Genauso gut ist es möglich, diejenige Kombination von Komponenten und Parametern zu suchen, für die der Energieverbrauch einer Anlage minimal wird. In diesem Fall ist es jedoch notwendig, dass die Berechnung der Energie innerhalb des Modells durchgeführt wird, da ansonsten eine Bewertung der Komponenten in der Kostenfunktion nicht durchgeführt werden kann. Die Wahl der Kostenfunktion hängt folglich davon ab, welches Optimierungsziel verfolgt wird.

Multikriterielle Optimierungsprobleme Die bisherige Betrachtung geht von der Verwendung lediglich eines Optimierungszieles aus. In der Praxis ist es jedoch häufig wünschenswert, eine bezüglich mehrerer Optimierungsziele optimale Lösung zu erhalten. Diese Optimierungsziele können sich möglicherweise sogar gegenseitig widersprechen. Ein widersprüchliches Ziel ist es beispielsweise, ein Produkt möglichst schnell und mit gleichzeitig minimalen Kosten herzustellen. Derartige Optimierungsprobleme können auf zwei Arten gelöst werden, die nachfolgend erläutert werden.

Die naheliegendste Möglichkeit ist es, die einzelnen Kostenfunktionen additiv zu einer Gesamtkostenfunktion zu verknüpfen. Dies reduziert das mehrkriterielle Optimierungsproblem zumindest mathematisch auf ein gewöhnliches Optimierungsproblem mit einer einzigen Kostenfunktion. Mit Hilfe von Faktoren, mit denen die einzelnen Kostenfunktionen multipliziert werden können, kann eine Gewichtung der einzelnen Funktionen vorgenommen werden. Die Gesamtkostenfunktion lässt sich mathematisch schreiben als

$$f(\mathbf{x}, \mathbf{y}) = \lambda_1 \cdot f_1(\mathbf{x}, \mathbf{y}) + \lambda_2 \cdot f_2(\mathbf{x}, \mathbf{y}) + \cdots + \lambda_n \cdot f_n(\mathbf{x}, \mathbf{y}). \tag{3.3}$$

Hierbei bezeichnen f_1 bis f_n die Kostenfunktionen der einzelnen Optimierungsziele, λ_1 bis λ_n die Gewichtungsfaktoren und f die Gesamtkostenfunktion. Die Vektoren $\mathbf{x}$ sowie $\mathbf{y}$ beinhalten die reellwertigen sowie ganzzahligen Optimierungsvariablen. Der größte Nachteil dieser Vorgehensweise ist es, dass in den meisten Fällen die Gewichtungsfaktoren λ_1 bis λ_n nur schwierig festzulegen sind. Dies liegt zum einen daran, dass in der Regel die Größenordnungen der Funktionswerte der einzelnen Kostenfunktionen stark unterschiedlich sind. Daher muss zunächst der Faktor bestimmt werden, sodass die Einflüsse der einzelnen Kostenfunktionen auf die Gesamtkostenfunktion gleichwertig sind. Anschließend können die einzelnen Kostenfunktionen bei Bedarf zusätzlich gewichtet werden, sofern die Einflüsse der Kostenfunktionen auf die Gesamtkostenfunktion als unterschiedlich wichtig erachtet werden. Insgesamt ist die Festlegung in jedem Fall subjektiv, was dazu führt, dass das Ergebnis der Optimierung stark von der Person abhängt, die die Faktoren festlegt. Mit einem anderen

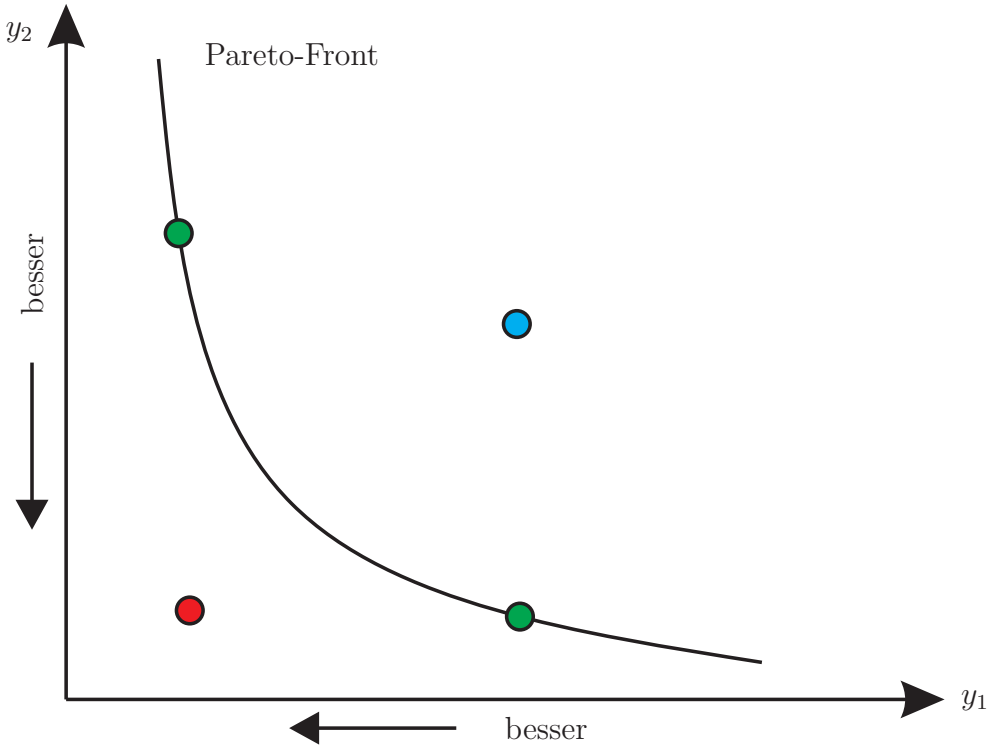

Abbildung 3.4: Graphische Darstellung des Zielfunktionsraumes mit Pareto-Front bei mehrkriterieller Optimierung [Menager (2012)]

Ansatz, der *Pareto-Optimierung*[1], werden die verschiedenen Optimierungsziele unabhängig voneinander betrachtet und die Kostenfunktionswerte getrennt ausgewertet. Diese Methode wird vor allem dann gewählt, wenn die einzelnen Kostenfunktionen nicht sinnvoll über Gewichtungsfaktoren kombiniert werden können, sodass das zuvor beschriebene Verfahren der Verknüpfung zu einer Gesamtkostenfunktion nicht angewendet werden kann. Im Vergleich zur eindimensionalen Optimierung ergibt sich durch die Pareto-Optimierung das so genannte *Pareto-Optimum* oder *Pareto-Set*. Dieses beinhaltet eine Menge von Lösungen, bei denen es nicht möglich ist, ein Optimierungsziel zu verbessern ohne gleichzeitig mindestens ein anderes zu verschlechtern.

Graphisch lässt sich das Pareto-Optimum durch eine $(n-1)$-dimensionale Hyper-Grenzfläche innerhalb des Zielfunktionsraumes darstellen. Im Falle eines zweidimensionalen Pareto-Optimierungsproblems ergibt sich eine eindimensionale Grenzlinie, welche üblicherweise die in Abbildung 3.4 dargestellte Form hat und auch als *Pareto-Front* bezeichnet wird. Die Entscheidung der Güte einer Lösung bezüglich einer anderen wird mit Hilfe der Pareto-Dominanz durchgeführt. Diese ist wie folgt definiert [Blum & Riedel (2004)]:

[1] nach Vilfredo Pareto, einem italienischen Ökonomen (1848–1923)

Definition 4 (Pareto-Dominanz)**.** Eine Lösung x^1 dominiert eine andere Lösung x^2 dann und nur dann, wenn sie in allen Zielfunktionen überlegen oder gleich ist und in mindestens einer Zielfunktion überlegen ist, d.h. es gilt

$$\begin{aligned} x^1 \succ x^2 \Leftrightarrow \forall\, i \in \{1,\ldots,m\} : f_i(x^1) \leq f_i(x^2) \\ \wedge \exists\, j \in \{1,\ldots,m\} : f_j(x^1) < f_j(x^2). \end{aligned}$$

Dominiert keine der beiden Lösung die andere, werden die Lösungen als *indifferent* bezeichnet. Das Pareto-Optimum ist die Menge aller dominanten Lösungen. Diese befinden sich graphisch auf der Pareto-Front und sind in Abbildung 3.4 grün einzeichnet. Die blaue Lösung ist dominiert und gehört daher nicht zum Pareto-Set, die rote Lösung verletzt die Randbedingungen und ist folglich nicht zulässig. Weitere Informationen zu Verfahren für die mehrkriterielle Optimierung finden sich in [Zitzler *et al.* (2004)] sowie [Menager (2012)].

3.2.1 Definition der Anforderungen

Für die erfolgreiche Umsetzung eines durchgängigen Engineerings in der Praxis, wie es in Abschnitt 3.1.3 beschrieben ist, wird ein Optimierungstool benötigt, welches auf Basis einer Modellbeschreibung des Systems automatisiert die optimale Kombination aus Komponenten und Modellparametern bestimmt. Das bei der automatisierten Auslegung auftretende Optimierungsproblem kann mithilfe mathematischer Optimierungsverfahren, wie sie in Kapitel 2.3 vorgestellt wurden, gelöst werden. Damit ein durchgängiges Engineering in der Praxis umgesetzt werden kann, ist die Verwendung von offenen Standards aus den bereits in Abschnitt 1.2 genannten Gründen essentiell. Da ein derartiges Optimierungstool in der Praxis bisher nicht existiert, ist die Entwicklung eines solchen Tools ein wesentliches Ziel dieser Arbeit. Damit dieses unabhängig von kommerziellen Tools verwendet werden kann ist eine wesentliche Anforderung, das Optimierungstool auf Basis von offenen Standards umzusetzen.

Für die Erstellung der Modelle wird, aus den in Kapitel 2.1 dargestellten Gründen, die Modellierungssprache *Modelica* verwendet. Um ausführbaren Code aus dem Modelica-Modell zu erzeugen, wird der quelloffene *OpenModelica*-Compiler verwendet. Im Rahmen des OpenModelica Projektes wurde bereits das Tool *OMOptim* entwickelt, welches für eine Optimierung verwendet werden kann [Thieriot *et al.* (2011)]. Dieses Tool ermöglicht zwar die Verwendung sowohl reellwertiger als auch ganzzahliger Optimierungsvariablen, jedoch können Optimierungsprobleme, die auf Strukturänderungen des betrachteten Modells führen, nicht gelöst werden. Auch über die Software OMOptim hinaus ist derzeit kein freies Tool erhältlich, mit dem derartige Optimierungsprobleme auf Basis von Modellbeschreibungen in einem offenen Standard behandelt werden können. Gemischt-ganzzahlige Optimierungsprobleme werden zumeist mit heuristischen Verfahren gelöst, da deren Berechnung in

der Regel $\mathcal{NP}$-schwer ist und das Problem somit nicht in polynomieller Zeit gelöst werden kann. Die Zeit zur Lösung derartiger Probleme steigt exponentiell mit der Problemdimension [Kallrath (2013)]. Obwohl zumindest theoretisch einige deterministische Verfahren existieren, die die optimale Lösung des Problems garantieren, können diese aufgrund der benötigten Zeit in der Regel nicht eingesetzt werden. Für weitere Informationen zu deterministischen Verfahren zur Lösung gemischtganzzahliger Optimierungsprobleme sei auf [Thiel (2005)] und [Burer & Letchford (2012)] verwiesen.

Eine wichtige Anforderung für die Entwicklung des Optimierungsmoduls ist es daher, dass für die Optimierung unterschiedlicher Optimierungsprobleme, beispielsweise rein reellwertige oder gemischt-ganzzahlige Optimierungsprobleme, jeweils geeignete Verfahren zur Verfügung gestellt werden. Auch die Lösung mehrkriterieller Optimierungsprobleme soll in dem Framework ermöglicht werden. Das Simulationsmodell, welches als Grundlage der Optimierung dient, soll im Sinne eines durchgängigen Engineerings möglichst unverändert bleiben und nicht erst aufwendig für die Optimierung angepasst werden müssen. Ferner ist zu berücksichtigen, dass die Bedienung des Optimierungsframeworks intuitiv gestaltet wird, sodass dieses ohne spezielles Expertenwissen verwendet werden kann. Für einen wirtschaftlichen Einsatz des entwickelten Optimierungstools ist außerdem die Zeit, die für das Auffinden der optimalen Lösung benötigt wird, zu berücksichtigen.

3.2.2 Umsetzung der automatisierten Optimierung

Ein Optimierungstool zur automatisierten Auslegung von Systemen besteht aus zwei grundlegenden Komponenten, die während der Optimierung ständig Daten austauschen: dem eigentlichen Optimierungsalgorithmus sowie einer Kostenfunktion. Der Optimierungsalgorithmus berechnet auf Basis des aktuellen Kostenfunktionswertes einen neuen Parametersatz für die freien Optimierungsvariablen. Im Fall des in Abschnitt 2.3.1.2 vorgestellten Nelder-Mead-Verfahrens wird in diesem Schritt das Simplex verändert, während bei Verwendung des Genetischen Algorithmus, vgl. Abschnitt 2.3.2.1, die einzelnen Individuen der Population durch die Operatoren (Selektion, Rekombination sowie Mutation) modifiziert werden. In beiden Fällen entstehen nach Durchlaufen des Algorithmus neue Werte für die Optimierungsvariablen. Die Kostenfunktion dient dazu, die Güte der vom Optimierer berechneten Werte zu beurteilen. Die Kostenfunktion ist eine mathematische Beschreibung des Optimierungszieles. Die in Gleichung 3.1 dargestellte Abweichung zwischen Ist-Verhalten und vorgegebenem Soll-Verhalten ist ein Beispiel für eine Kostenfunktion. Das Optimierungsziel ist es, das Ist-Verhalten bestmöglich dem Soll-Verhalten anzunähern. Um das Ist-Verhalten zu ermitteln, ist das System mit den entsprechenden Werten für die Optimierungsvariablen zu simulieren. Es muss folglich für jede Auswertung der Kostenfunktion eine Simulation durchgeführt werden.

Um anstelle einer einzelnen Simulation eines Modelica-Modells eine Optimierung auf diesem durchzuführen, muss der Simulationskern um ein entsprechendes Op-

timierungsmodul erweitert werden. Zudem ist es notwendig, die Codegenerierung entsprechend anzupassen. Für die Lösung von gemischt-ganzzahligen Optimierungsproblemen wird der Genetische Algorithmus verwendet. In der Standardimplementierung des Verfahrens sind die Operatoren (Rekombination, Mutation) nicht für die im Rahmen dieser Arbeit zu lösenden Optimierungsprobleme, also die automatische Bestimmung der optimalen Kombination aus reellwertigen Parametern sowie Komponenten, geeignet. Diese Problematik wird im Abschnitt 3.2.2.1 eingehender beleuchtet. Anschließend werden spezielle Anpassungen der Standard-Operatoren des Genetischen Algorithmus entwickelt.

Eine Schwierigkeit, die bei der Verwendung von Modelica-Modellen zur Optimierung auftritt, ist die Realisierung von strukturellen Änderungen innerhalb des Modells, wie sie beim Komponententausch auftreten. Diese Art von Optimierungsproblem spielt bei der automatisierten Auslegung von Systemen eine große Rolle. Eine Lösung derartiger Optimierungsprobleme wird derzeit von keinem Tool unterstützt. Die Problematik wird im Abschnitt 3.2.2.2 ausführlich diskutiert. Im Rahmen dieser Arbeit wird ein Ansatz entwickelt, wie Strukturänderungen innerhalb des Optimierungsproblems berücksichtigt werden können. Die Struktur des entwickelten Optimierungsmoduls wird in Abschnitt 3.2.2.3 beschrieben.

3.2.2.1 Anpassung der Standardimplementierung des Genetischen Algorithmus bei Verwendung ganzzahliger Optimierungsvariablen

Die Verwendung von ganzzahligen Optimierungsvariablen innerhalb eines Genetischen Algorithmus erfordert spezielle Operatoren für die Rekombination und Mutation. Die naheliegendste Implementierung des Rekombinations-Operators für ganzzahlige Optimierungsparameter ist die Bildung des Mittelwertes, wobei dieser anschließend, wenn nötig, auf eine ganze Zahl gerundet wird. Dies führt jedoch zu Problemen, sofern die Zahlen lediglich stellvertretend für spezielle Komponenten stehen, wie es bei der Komponentenoptimierung der Fall ist. Besteht die Optimierungsaufgabe darin, ein bezüglich einer vorgegebenen Kostenfunktion optimales Ventil für eine hydraulische Anlage zu ermitteln, wird die Komponente *Ventil* innerhalb des Optimierungsproblems als ganzzahlige Optimierungsvariable betrachtet. Der Zahl, die ein konkretes Ventil repräsentiert, können jedoch keine Informationen zu dem Ventil entnommen werden. Die Mittelwertbildung auf Basis der Zahl ist in diesem

Zahl	Ventil
1	4WRAE 6 E 07
2	4WRKE 35 E 1000L
3	4WREE 10 E 25
4	4WRAE 6 E 16

Tabelle 3.1: Auszug aus der Zuordnungstabelle der für die Optimierung verwendeten ganzen Zahlen und den repräsentierten Ventilen

Fall nicht zielführend. Das wird besonders an einem Beispiel deutlich. Die für das Beispiel verwendete Zuordnung zwischen Zahl und der repräsentierten Komponente kann Tabelle 3.1 entnommen werden. Die verwendeten Ventilbezeichnungen folgen dem bei der Bosch Rexroth AG verwendeten Schema. In dem Namen sind vier wesentliche Informationen enthalten:

- Ventilklasse (z.B. 4WRAE, 4WREE, ...)
- Nenngröße des Ventils (z.B. 10, 16, ...)
- Kolbentyp (z.B. E, E1, V, ...)
- Durchflusskennziffer (z.B. 15, 30, ...).

Besitzen die beiden Elternindividuen, die für eine Rekombination ausgewählt wurden, die Werte <1> sowie <3> für die Optimierungsvariable, wird durch die Rekombination bei Anwendung der Mittelwertbildung ein Nachkomme mit dem Wert <2> erzeugt. Bei den Ventilen, die durch die Zahlen 1 und 3 repräsentiert werden, handelt es sich um relativ kleine Ventile mit den Nenngrößen 6 bzw. 10 und den Durchflussmengen von 7 l/min bzw. 10 l/min. Die Aufgabe der Rekombination ist es, die Informationen der Eltern, die bereits erfolgsversprechende Werte besitzen und daher durch die Selektion ausgewählt wurden, zu verwenden und durch geschickte Kombination eine weitere Verbesserung zu erzielen. In diesem Fall entsteht jedoch, ausgedrückt durch die Zahl <2>, ein sehr großes Ventil mit der Nenngröße 35 und der Durchflusskennziffer 1000 l/min. Der auf diese Weise erzeugte Nachkomme beinhaltet keinerlei Informationen der Eltern und ist daher nutzlos.

Für eine Komponentenoptimierung sind die Standard-Operatoren des Genetischen Algorithmus folglich nicht ausreichend. Abhilfe kann geschaffen werden, wenn die Rekombination nicht auf Basis der Zahl erfolgt, sondern stattdessen die Informationen, die in dem Ventilnamen stecken, verwendet werden. Dazu können die vier Informationen, die in dem Ventilnamen stecken, einzeln verarbeitet werden. Es kann jeweils versucht werden, ein *„Mittelwert“* aus den beiden Elterninformationen zu generieren. Für die Nenngröße und die Durchflusskennziffer wird ein Wert gewählt, der zwischen den Werten der Eltern liegt. Die Ventilklasse und der Kolbentyp wird zufallsbasiert von einem der beiden Eltern übernommen. Dadurch ist eine Rekombination gewährleistet, die die Informationen der Eltern verwendet und basierend darauf einen sinnvollen Nachkommen erzeugt. Dies ist in [Menager (2013)] beschrieben. Für das hier gewählte Beispiel ist das Ventil, welches durch die Zahl <4> repräsentiert wird, eine sinnvolle Rekombination. Die Modifikation der Operatoren ist im Rahmen dieser Arbeit in den Genetischen Algorithmus implementiert worden.

3.2.2.2 Strukturelle Änderungen beim Komponententausch

Besteht die Optimierungsaufgabe darin, eine für die Realisierung eines vorgegebenen Systemverhaltens optimale Kombination aus Systemkomponente und Regler-

parameter auszuwählen, sind die zu variierende Systemkomponente und der Reglerparameter als freie Optimierungsvariablen zu wählen. Diese werden während der Optimierung vom Algorithmus verändert. Für die Bewertung der Güte der vom Algorithmus bestimmten konkreten Komponente und dem Wert für den Reglerparameter ist die Auswertung der Kostenfunktion erforderlich. Dafür muss eine Simulation des Modells mit dieser Komponente und dem Reglerparameter durchgeführt werden. Für die reellwertige Optimierungsvariable, den Reglerparameter, ist dazu lediglich der Wert des Parameters im Modell zu verändern. Eine Veränderung des Wertes für die ganzzahlige Optimierungsvariable, die die Systemkomponente repräsentiert, bedeutet jedoch, dass sich die Struktur des Modells verändert. Anstelle der vorher im System vorhandenen Komponente ist eine abweichende Komponente zu verwenden. Ein automatischer Austausch von Komponenten ist selbstverständlich nur dann möglich, wenn die Komponenten die gleichen Eingänge und Ausgänge besitzen[1].

Dies hat zur Folge, dass der in Abbildung 2.1 dargestellte Ablauf zur Übersetzung des Modells erneut durchlaufen werden muss. Der kompilierte Code des Modells muss anschließend in den bestehenden Simulator integriert werden. Der Austausch der Komponente muss weiterhin automatisiert innerhalb des Optimierungsalgorithmus erfolgen. Dafür besitzt der OpenModelica-Compiler eine CORBA-Schnittstelle[2], die einen automatisierten Aufruf des OpenModelica-Compilers ermöglicht. Diese Schnittstelle bietet jedoch keine Methode, die ein automatisiertes Austauschen der Komponente innerhalb des Modelica-Modells erlaubt. Das automatisierte Austauschen der Komponente muss folglich bereits modellseitig implementiert werden, sodass beim Übersetzen des Modells alle benötigten Informationen vollständig in den Simulator integriert sind.

Dafür wurde im Rahmen dieser Arbeit eine allgemeine Optimierungskomponente in Form eines Modelica-Modells entwickelt, die die zu optimierende Komponente im Simulationsmodell ersetzt. Eine derartige Optimierungskomponente wird für jeden Komponententyp, der optimiert werden soll, benötigt (z.B. Ventil, Zylinder, Pumpe). Die Funktionsweise ist jedoch in jedem Fall identisch. Die Optimierungskomponente dient als Schnittstelle zu den möglichen, innerhalb des Optimierungsproblems zulässigen Komponenten. Mit Hilfe eines Modellparameters kann zwischen den einzelnen hinterlegten Alternativen umgeschaltet werden, sodass sich stets eine bestimmte Komponente ergibt. Auf diese Weise muss für den Komponententausch ebenfalls lediglich ein Parameter verändert werden. Da jede Optimierungskomponente nur eine spezielle Klasse von Komponenten beinhaltet, die jeweils die selben Schnittstellen besitzt, ist der Komponententausch problemlos möglich. Die Ver-

[1] Es ist beispielsweise nicht möglich, ein direktgesteuertes Ventil durch ein vorgesteuertes Ventil zu ersetzen, da dieses weitere Eingänge für die Versorgung der Pilotstufe benötigt, die im Modell in dem Fall aber nicht vorhanden sind.

[2] Die *Common Object Request Broker Architecture* ist eine Spezifikation für eine objektorientierte Middleware für die Erstellung verteilter Anwendungen.

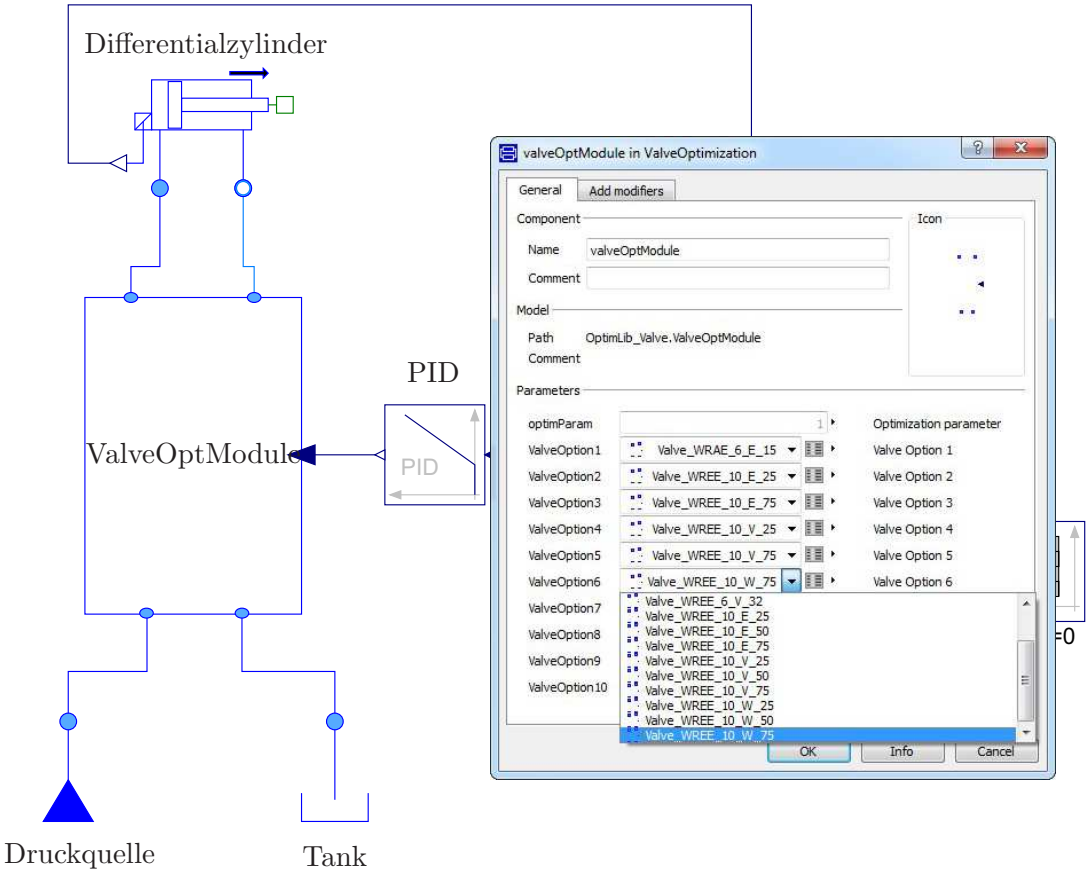

Abbildung 3.5: Verwendung der Komponente ValveOptModule aus der OptimLib innerhalb eines Modells

wendung geeigneter Modelica-Sprachmittel ermöglicht es, innerhalb des Parameterfensters graphisch diejenigen Komponenten auszuwählen, die innerhalb des Optimierungsproblems zulässig sind. Die Verwendung der Optimierungskomponente für Ventile innerhalb eines Modelica-Modells ist in Abbildung 3.5 dargestellt.

3.2.2.3 Struktur des Optimierungsmoduls

Ein wesentliches Ziel bei der Integration des Optimierungsmoduls in das OpenModelica-Projekt ist, dass dieses möglichst entkoppelt von dem Simulationskern ist. Vielmehr wird der Simulationskern lediglich bei der Auswertung der Kostenfunktion, die eine Simulation des Modells erfordert, aufgerufen. Das Optimierungsmodul selbst besteht aus drei Komponenten. Zunächst wird eine Konfigurationsklasse `OptimConfiguration` benötigt, die für die Konfiguration der Optimierung verwendet wird. Die Konfiguration legt unter anderem fest, welches Optimierungsverfahren verwendet wird und welches die freien Optimierungsvariablen sind. Weiterhin ist es möglich, das jeweilige Verfahren zu parametrieren. Detaillierte Informationen zur Parametrierung des Optimierungsmoduls befinden sich in Anhang A.1. Funktionen

zur Durchführung und Überwachung der Optimierung sind innerhalb der Klasse `OptimController` implementiert. Diese beinhaltet die mathematischen Optimierungsverfahren und ist für die Berechnung von numerischen Werten für die freien Optimierungsparameter zuständig. Die Kostenfunktion, die zur Berechnung eines Kostenfunktionswertes zur Beurteilung der Güte der für die freien Optimierungsparameter ermittelten Werte benötigt wird, ist in der Klasse `OptimCostfunction` implementiert. Ein Schema der Funktionsweise des Optimierungsmoduls ist in Abbildung 3.6 dargestellt. In dieser Grafik ist die Einbettung des bisherigen Simulationskerns in die Optimierung ersichtlich. Zur Berechnung des Kostenfunktionswertes ist die Durchführung einer Simulation mit den entsprechenden Werten für die Optimierungsparameter notwendig. Die Simulation wird unmittelbar aus der Kostenfunktion heraus aufgerufen.

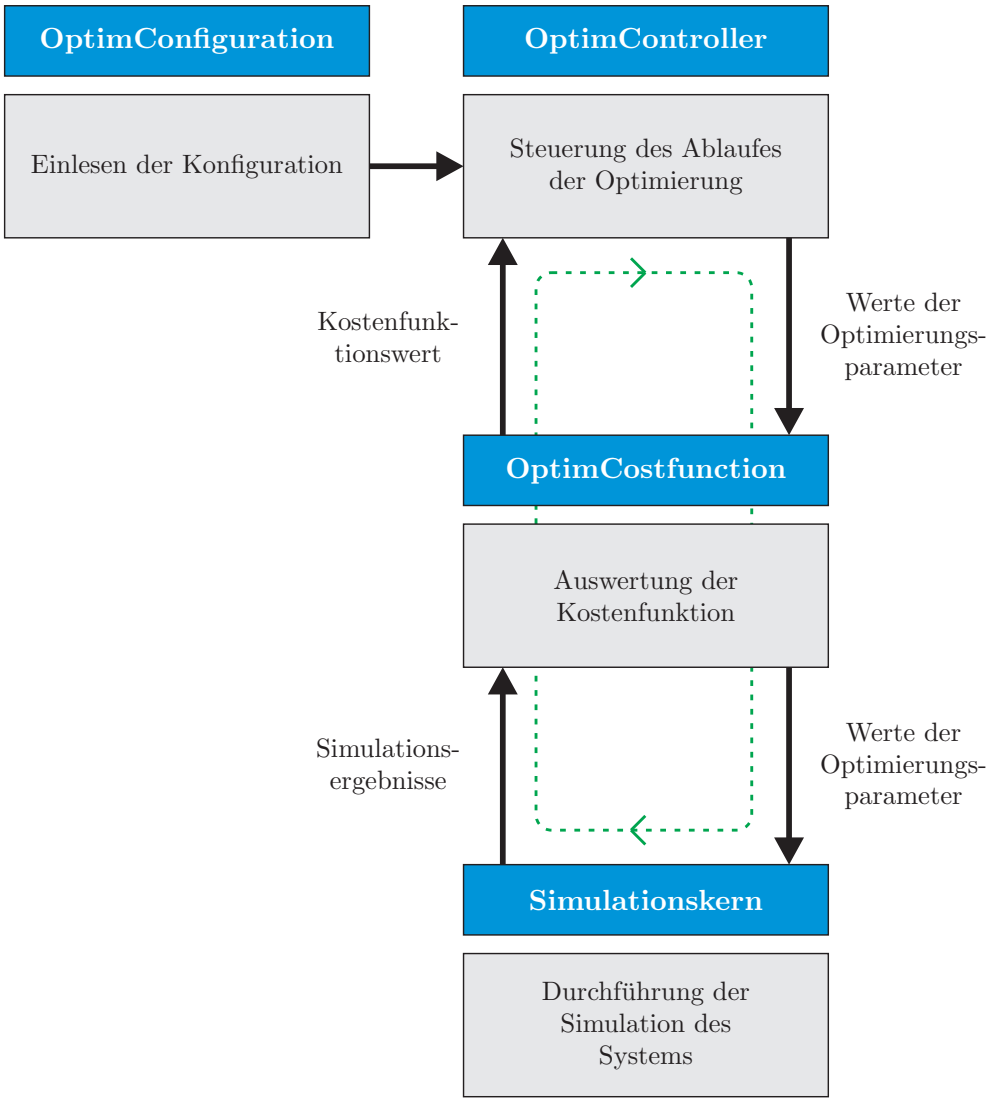

Abbildung 3.6: Struktur des Optimierungsmoduls

3.2.2.4 Verwendete Software-Bibliotheken

Zur Lösung des Optimierungsproblems werden verschiedene Verfahren in das Optimierungsmodul integriert. Im Kontext der automatisierten Auslegung von Systemen treten in der Regel Kostenfunktionen auf, für die die Berechnung des Gradienten mit sehr viel Aufwand verbunden ist, da sie nicht durch eine mathematische Funktion beschrieben werden kann. Die Kostenfunktion

$$\min_{x,y} \sum_{i=1}^{N} (y_i^{sim}(t, \mathbf{x}, \mathbf{y}) - y_i^{soll}(t))^2 \tag{3.4}$$

ist ein Beispiel für eine solche Funktion, da es sich um die Berechnung der Abweichung zwischen einer vorgegebenen Referenztrajektorie und diskreten Datenpunkten aus der Simulation handelt. Die Verwendung beispielsweise des Gradientenverfahrens aus Abschnitt 2.3.1.1 ist folglich nicht zielführend. Stattdessen muss auf spezielle ableitungsfreie Optimierungsverfahren zurückgegriffen werden.

In Kapitel 2.3.1.2 wurde das ableitungsfreie Nelder-Mead-Verfahren vorgestellt. Dieses eignet sich für die Optimierung rein reellwertiger Optimierungsprobleme ohne Nebenbedingungen, wie sie beispielsweise bei einer reinen Parameteroptimierung des Reglers auftreten. Für die Verwendung innerhalb dieser Arbeit wird eine frei erhältliche C++-Implementierung des Nelder-Mead-Verfahrens verwendet. Die originale Implementierung ist von O'Neill in FORTRAN entwickelt worden [O'Neill (1971)]. Später wurde die Implementierung von John Burkardt nach C++ portiert. Diese Version trägt den Namen *ASA047*[1].

Ein wesentlicher Fokus in dieser Arbeit liegt auf der Optimierung von gesamten Systemen mit der Suche nach optimalen Systemkomponenten, was auf gemischtganzzahlige Optimierungsprobleme führt. Für die Lösung derartiger Probleme sind heuristische Verfahren wie der Genetische Algorithmus geeignet. Eine Software-Bibliothek, die eine Vielzahl unterschiedlicher evolutionärer Algorithmen beinhaltet, ist die TEA[2]-Toolbox (Toolbox for Evolutionary Algorithms) der Universität Dortmund [Emmerich & Hosenberg (2001)]. Diese Bibliothek stellt ebenfalls Verfahren für die *multikriterielle* Optimierung bereit. Wie bereits zuvor in Abschnitt 3.2.2.1 beschrieben, sind die Standard-Operatoren des Genetischen Algorithmus nicht ausreichend zur effizienten Lösung der Optimierungsprobleme. Ein wesentlicher Aspekt bei der Entwicklung des Optimierungsmoduls ist es, dass eine effiziente Lösung der im Rahmen der automatisierten Auslegung von technischen Systemen auftretenden Optimierungsproblemen realisiert wird. Die in dem Abschnitt vorgestellten Modifikationen der Operatoren sind daher im Rahmen dieser Arbeit in die TEA-Toolbox integriert worden.

[1] *Nelder-Mead Implementierung:* `http://people.sc.fsu.edu/~jburkardt/cpp_src/asa047/asa047.html`

[2] *Genetischer Algorithmus Implementierung:* `http://sfbci.cs.uni-dortmund.de/index.php?option=com_content&task=view&id=28&Itemid=160`

Eine weitere Klasse von Optimierungsverfahren, die für die Lösung gemischt-ganzzahliger Optimierungsprobleme eingesetzt werden kann, ist die Partikelschwarmoptimierung. In dieser Arbeit wird die von Kyriakos Kentzoglanakis entwickelte C-Implementierung eines PSO-Verfahrens[1] verwendet. Diese Implementierung beinhaltet den originalen, von Eberhart und Kennedy vorgestellten Algorithmus. Zusätzlich sind weitere Nachbarschaftsstrategien implementiert. In der Standardversion ist dieser Algorithmus nicht für gemischt-ganzzahlige Optimierungsprobleme, sondern lediglich für rein reellwertige Optimierungsprobleme einsetzbar. Aus diesem Grund wird der Algorithmus im Rahmen dieser Arbeit durch geeignete Modifikationen für derartige Probleme angepasst. Dazu wird die in Abschnitt 2.3.2.2 vorgestellte Erweiterung des ursprünglichen Algorithmus integriert, bei der die Werte der diskreten Optimierungsvariablen zunächst als reellwertige Variablen verarbeitet werden und anschließend auf eine ganze Zahl gerundet werden.

3.2.2.5 Parallelisierung des Genetischen Algorithmus

Die Durchführung einer Optimierung kann für komplexe Systeme mit mehreren freien Optimierungsvariablen sehr zeitintensiv sein. Dies liegt unter anderem daran, dass bei Verwendung des Genetischen Algorithmus für das Auffinden einer Lösung nahe des globalen Optimums viele Iterationen und eine hohe Populationsgröße notwendig sind. Nachdem zu Beginn einer Iteration in der Selektion geeignete Elternindividuen der vorherigen Generation ausgewählt wurden und diese durch eine Rekombination sowie eine Mutation Nachkommen erzeugt haben, muss für sämtliche Nachkommen der Fitnesswert bestimmt werden. An dieser Stelle ist zur Bestimmung eines jeden Fitnesswertes eine Simulation mit den entsprechenden Werten der freien Optimierungsparameter notwendig. Diese Simulationen werden heute zumeist nacheinander ausgeführt, obwohl keinerlei Abhängigkeiten zwischen den einzelnen auszuwertenden Individuen bestehen.

Wird berücksichtigt, dass in heutigen Computern in der Regel mehrere Prozessorkerne zur Verfügung stehen, ist es sinnvoll, die Berechnung der einzelnen Fitnesswerte bzw. die Durchführung der einzelnen Simulationen zu parallelisieren und auf die zur Verfügung stehenden Prozessorkerne zu verteilen. Der Genetische Algorithmus eignet sich aufgrund der Unabhängigkeit der einzelnen Individuen voneinander besonders gut für eine Parallelisierung. Da die Zeit, die für die Optimierung benötigt wird, im Wesentlichen durch die Dauer der einzelnen Simulationen verursacht wird, reduziert eine parallele Auswertung die Optimierungsdauer signifikant. Für die Parallelisierung innerhalb des Codes wird die offene Schnittstelle *OpenMP*[2] verwendet. Damit die Durchführung der Simulationen parallel erfolgen kann, muss das System innerhalb des Simulationskerns mehrfach geladen werden, da jede Instanz des Systems mit unterschiedlichen Werten für die freien Optimierungsvariablen initialisiert

[1] *PSO Implementierung:* `https://github.com/kkentzo/pso`
[2] Open Multi-Processing

wird. Die dafür notwendigen Anpassungen am Simulationskern sind im Rahmen der Arbeit in diesen integriert worden.

3.3 Toolchain zur Ausführung von Modellen auf Steuerungshardware

Nachdem das zu entwickelnde System unter Zuhilfenahme der im vorherigen Kapitel vorgestellten Optimierungsmethoden simulativ ausgelegt ist, besteht der nächste Schritt darin, den Regelungsalgorithmus unter Echtzeitbedingungen auf der Steuerungshardware, die letztendlich an der realen Anlage verwendet wird, zu testen. Die Implementierung des Reglercodes auf der Steuerungshardware erfolgt bisher zumeist in klassischen SPS-Programmiersprachen nach IEC 61131-3. Das während der Auslegung erstellte Simulationsmodell des Reglers wird in dieser Phase in der Regel nicht weiterverwendet. Stattdessen wird eine komplette Re-Implementierung des bestehenden Codes durchgeführt oder aber ein bereits auf der Steuerungshardware vorhandener, fest implementierter Standard-Reglercode verwendet, der für das spezielle, zu entwickelnde System möglicherweise nicht optimal ist.

Durch die Umsetzung eines durchgängigen, modellbasierten Engineerings gelingt es, die Re-Implementierung zu vermeiden. Dazu wird das Modell aus der Auslegungs- in die Inbetriebnahmephase überführt. Dieses Vorgehen führt neben deutlichen Zeiteinsparungen sowie damit einhergehenden Kosteneinsparungen zu weiteren Vorteilen. In diesem Fall kann der bereits entwickelte und virtuell am System getestete Regler verwendet werden, anstatt eine Re-Implementierung durchzuführen, die eine zusätzliche potentielle Fehlerquelle darstellt.

Eine wesentliche Methode des modellbasierten Engineerings zur Überführung von Modellen von einer Entwicklungsphase in die darauffolgende ist die Codegenerierung. Diese ermöglicht es, aus einem Simulationsmodell ausführbaren Code zu generieren, der anschließend auf der Steuerungshardware verwendet wird. Die Verwendung des Reglermodells auf der Steuerungshardware zur Anwendung des *Rapid Control Prototyping* stellt lediglich eine Anwendung dar, wie aus Modellen generierter Code auf Steuerungen zu einer Effizienzsteigerung sowohl innerhalb des Entwicklungsprozesses als auch in der Phase des Betriebs führen kann. Wie bereits in Abschnitt 1.1 ausführlich beschrieben, lassen sich Modelle der Regelstrecke auf der Steuerung unter anderem für eine modellbasierte Diagnose oder für eine modellprädiktive Regelung einsetzen. Ein wesentlicher Baustein zur Umsetzung eines durchgängigen, modellbasierten Engineerings ist, wie in Abschnitt 3.1.3 diskutiert, eine Toolchain, die die Erzeugung von ausführbarem Code aus einer bereits vorhandenen Modellbeschreibung ermöglicht, welcher anschließend auf der Zielhardware ausgeführt wird. Eine derartige Toolchain wird im Rahmen dieser Arbeit entwickelt.

Abbildung 3.7: IndraControl XM22 Industriesteuerung der Bosch Rexroth AG

Eine wesentliche Anforderung an die Toolchain ist die Verwendung offener Standards. Dies gewährleistet zum einen, dass die Toolchain aufgrund entfallender Lizenzkosten ebenso von kleinen und mittleren Unternehmen eingesetzt werden kann, die ansonsten von den Vorteilen modellbasierter Entwicklungsmethodiken nicht profitieren können. Zum anderen besteht auf diese Weise die Möglichkeit einer flexiblen Anpassung des generierten Codes auf spezielle Hardware, was bei Verwendung einer kommerziellen Codegenerierung nicht möglich ist. Diese sind in der Regel abgeschlossen und können daher nicht modifiziert werden, um sie für bestimmte Hardwareplattformen anzupassen. Die Toolchain basiert, wie bereits das im vorherigen Kapitel vorgestellte Optimierungstool, auf der standardisierten, freien Modellierungssprache Modelica. Dadurch wird ermöglicht, die bereits bei der Auslegung verwendeten Modelle, die auf Basis der Modellierungssprache Modelica erstellt wurden, weiterzuverwenden.

Im Rahmen dieser Arbeit werden hauptsächlich Industriesteuerungen des Typs *IndraControl XM22* von Bosch Rexroth als Hardwareplattform verwendet. Die Steuerung basiert auf einem Intel Atom E660T Prozessor mit 1300 MHz und besitzt 512 MB Arbeitsspeicher [Bosch Rexroth AG (2015b)]. Auf der Steuerung wird das Echtzeitbetriebssystem *VxWorks*[1] in der Version 6.9 verwendet. Abbildung 3.7 zeigt eine IndraControl XM22. Für rechenintensive Anwendungen, beispielsweise zur Regelung einer Vielzahl von Achsen, kommen anstelle klassischer Industriesteuerungen immer häufiger Industrie-PCs zum Einsatz. Ein Beispiel für eine solche Hardware stellt die *IndraControl VPB40.3* von Bosch Rexroth dar. Diese Steuerungshardware verwendet ebenfalls das Echtzeitbetriebssystem VxWorks, besitzt jedoch eine deutlich leistungsstärkere Hardware. Diese basiert auf einem Intel Core i7-620M

[1] `http://www.windriver.com/products/vxworks/`

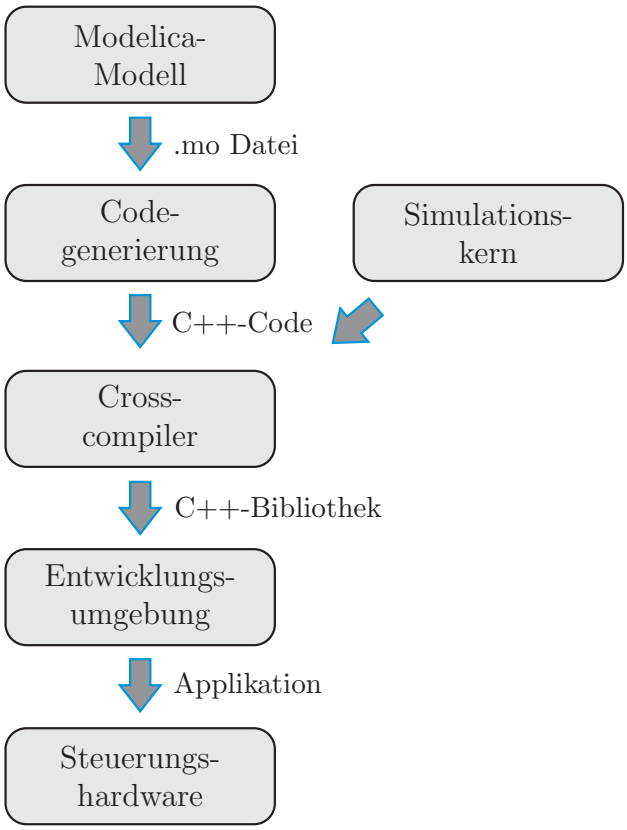

Abbildung 3.8: Struktur der Toolchain

Dual-Core-Prozessor mit 2,66 GHz Taktfrequenz pro Kern und 8 GB Arbeitsspeicher [Bosch Rexroth AG (2015a)].

Die Toolchain besteht aus mehreren Komponenten, die miteinander interagieren. Bei der Entwicklung der Toolchain ist es wichtig, dass diese allgemein gehalten und damit für beliebige Modelica-Modelle verwendbar ist. Abbildung 3.8 stellt die Struktur graphisch dar. Die einzelnen Komponenten werden sowohl nachfolgend als auch in [Menager *et al.* (2014a)] und [Menager *et al.* (2015b)] ausführlich beschrieben.

3.3.1 Codegenerierung aus Modelica-Modellen

Ausgangspunkt der Toolchain ist ein beliebiges Modelica-Modell. Da Modelica eine Modellierungs*sprache* ist, ist zur Übersetzung in ein lauffähiges Programm ein Compiler notwendig. Im Rahmen dieser Arbeit wird der freie *OpenModelica*-Compiler verwendet. Der Einsatz des OpenModelica-Compilers hat den Vorteil, dass dieser modular aufgebaut ist mit dem Ziel, eigene Module integrieren zu können. Der letzte Schritt innerhalb des Compilers, nachdem die Modellgleichungen im Frontend sowie Backend bearbeitet und optimiert wurden, ist die Erzeugung von Code (vgl. dazu Abbildung 2.1). Der Aufbau des Codegenerierungs-Moduls innerhalb des OpenModelica-Compilers ist templatebasiert. Mit Hilfe der speziell für den

OpenModelica-Compiler entwickelten Templatesprache Susan ist es möglich, verschiedene Codegenerierungs-Module zu entwickeln und in den Compiler zu integrieren. Dies erlaubt es, hardwarespezifischen Code für unterschiedliche Plattformen zu erzeugen, sofern dies benötigt wird. Standardmäßig wird C-Code oder C++-Code generiert.

Der templatebasierte Aufbau der Codegenerierung erlaubt es somit, auf komfortable Weise vorhandene Schnittstellen der verwendeten Steuerungshardware in den generierten Code zu integrieren. Im Rahmen dieser Arbeit wurde die vorhandene C++-Codegenerierung für die Echtzeitanwendung auf Industriesteuerungen erweitert. Durch die zusätzliche Angabe eines Arguments im Compiler-Aufruf wird festgelegt, ob Code für eine Offline-Simulation unter Windows oder eine Echtzeitsimulation auf der Steuerungshardware generiert wird. Auf welche Weise der Code in die Laufzeitumgebung der Steuerung integriert werden kann, wird in Abschnitt 3.3.3 beschrieben.

3.3.2 Anpassung des Simulationskerns für Echtzeitanwendungen

Wie bereits in Abschnitt 2.1.3.1 erläutert, ist zur Simulation eines Modells ein Simulationskern notwendig. Dieser steuert den Ablauf der Simulation und enthält die numerischen Verfahren zur Lösung der innerhalb des Modells auftretenden Gleichungen. Während einer Offline-Simulation besteht in der Regel kein Zusammenhang zwischen dem Voranschreiten der Simulationszeit und der realen Zeit. Die Simulation soll, unter Beachtung einer vorgegebenen Genauigkeit, stets so schnell wie möglich ablaufen. Um dieses Ziel zu erreichen, beinhalten die numerischen Verfahren Techniken wie eine *adaptive Schrittweitensteuerung*. Weiterhin wird die Jacobi-Matrix nur dann neu ausgewertet, wenn ein Schritt fehlgeschlagen ist. Die eingesetzten Techniken sorgen zwar dafür, dass der Simulationsaufwand insgesamt minimiert wird, der Aufwand pro Schritt schwankt dadurch jedoch sehr stark.

Die Anforderungen an eine Echtzeitsimulation unterscheiden sich gravierend von denen an eine Offline-Simulation. Bei einer Echtzeitsimulation ist es zwingend notwendig, dass Simulationszeit und reale Zeit synchron zueinander sind. Daher muss garantiert werden, dass die Berechnung eines Integrationsschrittes innerhalb einer vorgegebenen Zeitspanne durchgeführt wird. Diese vorgegebene Zeitspanne ist der Echtzeittakt. Dies ist eine harte Einschränkung für die zulässigen numerischen Verfahren, da stets mit der ungünstigsten Laufzeit (kleinste benötigte Schrittweite) kalkuliert werden muss. Dies macht die zuvor vorgestellte Technik der adaptiven Schrittweitensteuerung bei der Echtzeitsimulation überflüssig, da in diesem Fall die kleinste benötigte Schrittweite zur Steigerung der Genauigkeit der Lösung in jedem Zeitschritt verwendet werden kann. Daher müssen Verfahren verwendet werden, deren Aufwand eine obere Schranke besitzt, insbesondere ist die Verwendung von iterativen Elementen grundsätzlich nicht zulässig, da für diese keine maximale Laufzeit angegeben werden kann, bis die Lösung konvergiert. Ein echtzeitfähiges

numerisches Lösungverfahren ist daher ein Verfahren mit fester Schrittweite und ohne iterative Komponenten. Der Aufwand pro Schritt sollte möglichst gleichmäßig und so gering wie möglich sein [Kampfmann (2014)].

Der Ablauf eines Simulationsschrittes ist grundsätzlich bei allen Lösungsverfahren ähnlich. Dieser ist in Abbildung 3.9 dargestellt. Zunächst wird die Differentialgleichung integriert. Dazu ist, abhängig vom verwendeten Verfahren, mindestens eine Funktionsauswertung und gegebenenfalls auch eine Auswertung der Jacobi-Matrix erforderlich. Anschließend wird überprüft, ob während des Integrationsschrittes eine Unstetigkeit, auch als *Event* bezeichnet, aufgetreten ist. Während einer Simulation können zwei verschiedene Arten von Unstetigkeiten auftreten: zeitabhängige Unstetigkeiten (Time-Events) sowie zustandsabhängige Unstetigkeiten (State-Events). Während zeitabhängige Events bereits vor Simulationsstart behandelt werden können, müssen zustandsabhängige Events zwingend während der Simulation behandelt werden, da der Zeitpunkt des Auftretens des Events vor Simulationsstart nicht bekannt ist. Sind in dem Simulationsschritt keine Events auf-

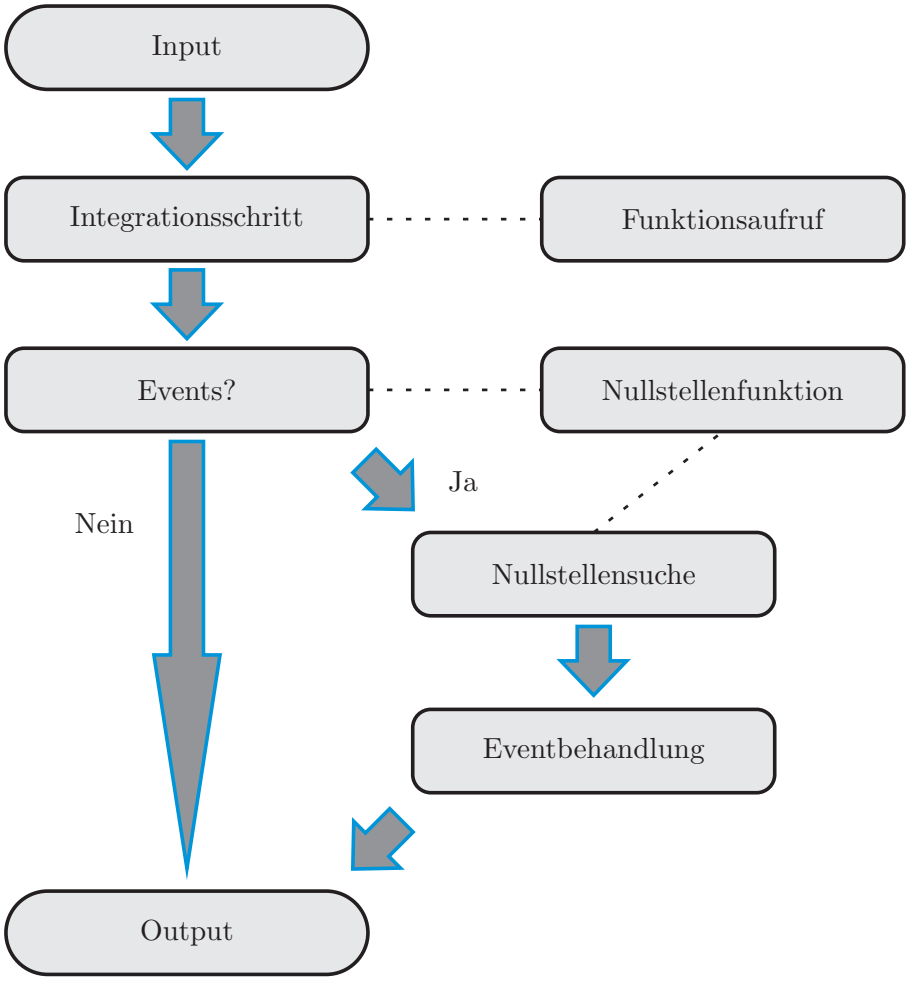

Abbildung 3.9: Ablauf eines Simulationsschrittes [Kampfmann (2014)]

getreten, ist dieser beendet. Ist hingegen ein State-Event aufgetreten, muss dieses vor Beendigung des Simulationsschrittes behandelt werden. Die Behandlung von State-Events ist vor allem bei Echtzeitsimulationen problematisch, da im Rahmen der Eventbehandlung unter Umständen weitere Integrationsschritte notwendig sind, um am Ende des Simulationsschrittes synchron zur Echtzeit zu sein.

Die beiden Typen von Events sowie das Auftreten von algebraischen Schleifen werden in den nachfolgenden Abschnitten 3.3.2.2 bis 3.3.2.4, insbesondere im Kontext der Verwendung innerhalb einer Echtzeitsimulation, ausführlich erläutert. Wie bereits erwähnt, werden für die Echtzeitsimulation zur Durchführung des Integrationsschrittes spezielle numerische Lösungsverfahren benötigt. Daher werden im nächsten Abschnitt zunächst verbreitete Lösungsverfahren auf ihre Echtzeitfähigkeit überprüft. Anschließend werden spezielle echtzeitfähige Verfahren behandelt. Die benötigten Erweiterungen wurden im Rahmen der Arbeit in den C++-Simulationskern des OpenModelica-Projektes integriert.

3.3.2.1 Numerische Verfahren für die Echtzeitsimulation

Für die Simulation von Modellen unter Berücksichtigung von Echtzeitanforderungen werden spezielle echtzeitfähige Integrationsverfahren benötigt. Nachdem die Gleichungen des Modelica-Modells, die zunächst ein großes differential-algebraisches Gleichungssystem ergeben, das Frontend und Backend des OpenModelica-Compilers durchlaufen haben, liegen sie als Differentialgleichungssystem erster Ordnung vor. Wie bereits in Abschnitt 2.2 ausführlich beschrieben, stehen zur Lösung von Differentialgleichungen eine Vielzahl numerischer Methoden zur Verfügung. Die Forderung nach Echtzeitfähigkeit schränkt die Auswahl jedoch deutlich ein. Ein echtzeitfähiges Integrationsverfahren wurde bereits zuvor als Verfahren mit fester Schrittweite und ohne iterative Komponenten charakterisiert. Im Folgenden werden zunächst die in Abschnitt 2.2 beschriebenen Verfahren auf ihre Echtzeitfähigkeit untersucht und im Anschluss weitere, spezielle echtzeitfähige Verfahren vorgestellt.

Bewertung klassischer Verfahren im Hinblick auf Echtzeitfähigkeit Das in Abschnitt 2.2.1.1 beschriebene Explizite Euler-Verfahren ist sehr einfach zu implementieren und benötigt aufgrund der Tatsache, dass es sich um ein explizites Verfahren handelt und somit keine iterativen Elemente notwendig sind, um die Gleichung auszuwerten, einen konstanten Aufwand. Es wird für die Durchführung eines Simulationsschrittes jeweils eine Funktionsauswertung mit einer konstanten Anzahl an arithmetischen Rechenoperationen benötigt. Weiterhin verwendet das Explizite Euler-Verfahren eine konstante Schrittweite, weswegen das Verfahren hervorragend für eine Echtzeitsimulation geeignet ist. Aufgrund der niedrigen Konsistenzordnung und der schlechten Stabilitätseigenschaften ist es jedoch nur begrenzt anwendbar. Besonders für mathematisch steife Systeme, die in der Praxis jedoch häufig auftreten, kann die Stabilität nur durch sehr kleine Schrittweiten gewährleistet werden, die bei einer Echtzeitsimulation weit unterhalb des notwendigen Echtzeittaktes liegen.

Das Implizite Euler-Verfahren, welches in Abschnitt 2.2.1.2 vorgestellt wurde, eignet sich aufgrund der guten Stabilitätseigenschaften auch für steife Systeme. Eine Anwendung unter Echtzeitbedingungen ist jedoch nicht möglich, da in jedem Simulationsschritt ein nichtlineares Gleichungssystem iterativ gelöst werden muss. Bei diesen Verfahren ist keine Vorhersage möglich, innerhalb welcher Zeitspanne die Lösung gefunden wird. Die Einhaltung einer vorgegebenen Zeit zur Berechnung der Lösung ist jedoch eine fundamentale Voraussetzung für den Echtzeitbetrieb.

Die anschließend in Abschnitt 2.2.2.1 beschriebenen Expliziten Runge-Kutta-Verfahren sind grundsätzlich sehr gut für Echtzeitsimulationen geeignet. Da die Verfahren explizit sind, ist der Aufwand der Verfahren konstant und bekannt, iterative Elemente sind nicht vorhanden. Jedoch sind die Expliziten Runge-Kutta-Verfahren, ebenso wie das Explizite Euler-Verfahren, nicht A-stabil. Die Wahl der Schrittweite erfolgt daher nicht nur anhand von Genauigkeitsanforderungen, sondern muss ebenfalls unter Berücksichtigung von Stabilitätsbetrachtungen erfolgen. Für mathematisch steife Systeme sind daher sämtliche Expliziten Runge-Kutta-Verfahren nicht geeignet. Zur Simulation von nicht-steifen Modellen wie Reglermodelle können die Verfahren sinnvoll eingesetzt werden. Da es sich bei Expliziten Runge-Kutta-Verfahren um Einschrittverfahren handelt, können diese Verfahren sehr gut mit Unstetigkeiten umgehen. Bei Einschrittverfahren kann jeder Simulationsschritt als Lösen eines neuen Anfangswertproblemes betrachtet werden. Die aktuelle Lösung wird als Anfangsbedingung für den nächsten Simulationsschritt verwendet. Treten während der Simulation eines Schrittes Unstetigkeiten auf, wird das System nach einer Lokalisierung der Unstetigkeit mit neuen Werten initialisiert. Der nachfolgende Simulationsschritt kann ebenso als ein neues Anfangswertproblem aufgefasst werden. Für das verwendete Explizite Runge-Kutta-Verfahren macht es daher keinen Unterschied, ob ein Event auftritt oder nicht, da der durchgeführte Schritt lediglich auf Basis des aktuellen Zustandes erfolgt.

Für die Impliziten Runge-Kutta-Verfahren aus Abschnitt 2.2.2.2 gilt die gleiche Argumentation wie für das Implizite Euler-Verfahren. Die Verfahren können aufgrund der guten Stabilitätseigenschaften in der Praxis auch für mathematisch steife Systeme eingesetzt werden. Allerdings enthalten sämtliche Verfahren dieser Klasse iterative Elemente, sodass die Verfahren dieser Klasse für eine Echtzeitsimulation grundsätzlich nicht geeignet sind.

Rosenbrock-Verfahren Die Verwendung von Expliziten Runge-Kutta-Verfahren bietet den Vorteil, dass der Aufwand pro Schritt konstant ist. Diese Verfahren sind allerdings nicht für steife Probleme geeignet. Implizite Runge-Kutta-Verfahren hingegen sind sehr gut für steife Probleme geeignet, jedoch ist ihr Aufwand pro Schritt nicht a priori bekannt. Das für die Echtzeitsimulation steifer Systeme verwendete Verfahren sollte also möglichst die Vorteile beider Verfahren vereinen und sowohl einen festen Aufwand pro Schritt als auch gute Stabilitätseigenschaften bieten.

Dies kann erreicht werden, wenn innerhalb der impliziten Verfahren die iterativen Anteile durch eine Linearisierung entfernt werden. Dieses Vorgehen wird am Beispiel des Impliziten Euler-Verfahrens demonstriert. Die Verfahrensvorschrift des Impliziten Euler-Verfahrens lautet, wie bereits in Abschnitt 2.2.1.2 beschrieben,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}). \tag{3.5}$$

Wird eine Taylorentwicklung der Funktion $\mathbf{f}$ um den Punkt $(\mathbf{x}_n,\, t_n)$ durchgeführt, ergibt sich

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\left[\mathbf{f}(\mathbf{x}_n, t_n) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_n, t_n) \cdot (\mathbf{x}_{n+1} - \mathbf{x}_n) + \frac{\partial \mathbf{f}}{\partial t}(\mathbf{x}_n, t_n) \cdot \underbrace{(t_{n+1} - t_n)}_{=h}\right]. \tag{3.6}$$

Ausmultiplizieren führt zu dem Ausdruck

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_n, t_n) + h\,\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_n, t_n) \cdot \mathbf{x}_{n+1} - h\,\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_n, t_n) \cdot \mathbf{x}_n + h^2\,\frac{\partial \mathbf{f}}{\partial t}(\mathbf{x}_n, t_n). \tag{3.7}$$

Wird $\mathbf{x}_{n+1}$ auf die linke Seite gebracht, ergibt sich

$$\left(\mathbf{I} - h\,\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_n, t_n)\right)\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_n) - h\,\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_n, t_n) \cdot \mathbf{x}_n + h^2\,\frac{\partial \mathbf{f}}{\partial t}(\mathbf{x}_n, t_n). \tag{3.8}$$

Dieses Verfahren wird auch als *Linear-Implizites Euler-Verfahren* bezeichnet und hat die Konsistenzordnung eins. Die Berechnung eines Schrittes besteht bei Verwendung dieses Verfahrens aus zwei Schritten. Zunächst ist die rechte Seite zu berechnen. Anschließend ist ein lineares Gleichungssystem zu lösen, um die Lösung $\mathbf{x}_{n+1}$ zu erhalten. Der Aufwand für die Berechnung eines Schrittes ist damit konstant, da sowohl das Auswerten der rechten Seite als auch das Lösen des linearen Gleichungssystems einen fixen Aufwand verursacht. Das Linear-Implizite Euler-Verfahren ist weiterhin A-stabil, da die Dahlquist'sche Testgleichung selbst linear ist, weswegen die Linearisierung von $\mathbf{f}$ in der Verfahrensvorschrift bezüglich $\mathbf{x}$ dieselbe Lösung liefert wie das Lösungsverfahren selbst.

Mit Hilfe des Linear-Impliziten Euler-Verfahrens ist es gelungen, die Vorteile des Expliziten Euler-Verfahrens und des Impliziten Euler-Verfahrens zu kombinieren. Das Verfahren ist A-stabil, enthält jedoch keine iterativen Elemente, womit die Auswertung einen fixen Aufwand besitzt. Aufgrund der niedrigen Konsistenzordnung besteht jedoch der Wunsch nach Verfahren mit einer höheren Ordnung. Allgemein gilt, dass bei der Linearisierung eines Impliziten Runge-Kutta-Verfahrens ein lineares Gleichungssystem der Dimension $n \cdot s$ gelöst werden muss. Dabei bezeichnet n die Dimension des Differentialgleichungssystems und s die Anzahl der Stufen des Runge-Kutta-Verfahrens. Da das Lösen eines linearen Gleichungssys-

tems mit dem Gauß-Verfahren den Aufwand $\mathcal{O}(n^3)$ besitzt, ist es wünschenswert, das große Gleichungssystem in mehrere kleine zu unterteilen. Dies gelingt mit Hilfe der Diagonal-Impliziten Runge-Kutta-Verfahren. In diesem Fall müssen lediglich s lineare Gleichungssysteme der Dimension n gelöst werden, was einen deutlich geringeren Aufwand darstellt.

Eine Linearisierung der Diagonal-Impliziten Runge-Kutta-Verfahren führt auf die Klasse der Rosenbrock-Wanner-Verfahren, häufig auch nur als Rosenbrock-Verfahren bezeichnet. Ein Verfahrensschritt eines s-stufigen Rosenbrock-Verfahrens ist durch

$$\mathbf{k}_i = h\,\mathbf{f}\Big(\mathbf{x}_n + \sum_{j=1}^{i-1} \alpha_{ij}\,\mathbf{k}_j,\ t_n + \alpha_i\,h\Big) + \gamma_i\,h^2\,\frac{\partial \mathbf{f}}{\partial t}\big(\mathbf{x}_n, t_n\big) + h\,\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\big(\mathbf{x}_n, t_n\big) \sum_{j=1}^{i} \gamma_{ij}\,\mathbf{k}_j \tag{3.9}$$

für alle $i = 1, ..., s$ sowie

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \sum_{j=1}^{s} b_j\,\mathbf{k}_j, \tag{3.10}$$

gegeben, wobei $\alpha_{ij}, \gamma_{ij}, b_i$ die bestimmenden Koeffizienten sind [Hairer & Wanner (1996)]. Die fehlenden Koeffizienten berechnen sich mit Hilfe der Gleichungen

$$\alpha_i = \sum_{j=1}^{i-1} \alpha_{ij} \qquad \gamma_i = \sum_{j=1}^{i} \gamma_{ij}. \tag{3.11}$$

In jeder Stufe eines Rosenbrock-Verfahrens ist ein lineares Gleichungssystem mit der Unbekannten $\mathbf{k}_i$ und der Systemmatrix

$$\mathbf{I} - h\,\gamma_{ii}\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\big(\mathbf{x}_n, t_n\big) \tag{3.12}$$

zu lösen. Der Aufwand zur Berechnung lässt sich jedoch deutlich reduzieren, wenn die Wahl der entsprechenden Koeffizienten geeignet erfolgt. Gilt beispielsweise

$$\gamma_{11} = \cdots = \gamma_{ss} = \gamma, \tag{3.13}$$

so ist die Systemmatrix aus Gleichung 3.12 in jeder Stufe identisch. Weiterhin ist es möglich, durch eine geschickte Wahl der Koeffizienten α_{ij} die Anzahl an notwendigen Funktionsauswertungen zur Berechnung der einzelnen Stufen in Gleichung 3.9 zu reduzieren. Gilt $\alpha_{(s-1)j} = \alpha_{sj}$ für $j = 1, \ldots, s-1$ sowie $\alpha_{ss} = 0$, sind die Funktionsargumente der letzten beiden Stufen identisch. Neben dem Lösen des linearen Gleichungssystems ist in jedem Schritt eine Matrix-Vektor-Multiplikation durchzuführen. Durch ein geschicktes Einführen neuer Variablen kann diese vermieden

werden [Hairer & Wanner (1996)]:

$$\mathbf{u}_i = \sum_{j=1}^{i} \gamma_{ij}\, \mathbf{k}_j, \qquad i = 1, \ldots, s. \tag{3.14}$$

Gilt weiterhin $\gamma_{ii} \neq 0$ für alle i, ist die Matrix $\mathbf{\Gamma} = (\gamma_{ij})$ invertierbar, sodass $\mathbf{k}_i$ aus $\mathbf{u}_i$ berechnet werden kann:

$$\mathbf{k}_i = \frac{1}{\gamma_{ii}}\, \mathbf{u}_i - \sum_{j=1}^{i-1} c_{ij}\, \mathbf{u}_j, \qquad \mathbf{C} = (c_{ij}) = \operatorname{diag}(\gamma_{11}^{-1}, \ldots, \gamma_{ss}^{-1}) - \mathbf{\Gamma}^{-1}. \tag{3.15}$$

Insgesamt lässt sich folgende Rechenvorschrift formulieren, die die Grundlage der im Rahmen dieser Arbeit implementierten Rosenbrock-Verfahren darstellt:

$$\begin{aligned}\left(\frac{1}{h\gamma}\mathbf{I} - \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_n, t_n)\right) \mathbf{u}_i = \mathbf{f}\left(\mathbf{x}_n + \sum_{j=1}^{i-1} a_{ij}\, \mathbf{u}_j,\ t_n + \alpha_i\, h\right) + \ldots \\ \ldots + \sum_{j=1}^{i-1} \left(\frac{c_{ij}}{h}\, \mathbf{u}_j\right) + \gamma h\, \frac{\partial \mathbf{f}}{\partial t}(\mathbf{x}_n, t_n)\end{aligned} \tag{3.16}$$

für alle $i = 1, \ldots, s$ sowie

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \sum_{j=1}^{s} m_j\, \mathbf{u}_j, \tag{3.17}$$

wobei $m_i, c_{ij}, a_{ij}, \gamma, \alpha_i$ Parameter sind [Hairer & Wanner (1996)]. Für die Parameter a_{ij} sowie m_i gelten darüber hinaus die Zusammenhänge

$$(a_{ij}) = (\alpha_{ij})\mathbf{\Gamma}^{-1} \qquad (m_1, \ldots, m_s) = (b_1, \ldots, b_s)\mathbf{\Gamma}^{-1}. \tag{3.18}$$

Insgesamt werden drei verschiedene Verfahren innerhalb des Simulationskerns zur Verfügung gestellt. Neben dem Linear-Impliziten Euler-Verfahren, im Rahmen dieser Arbeit auch als ROS1-Verfahren bezeichnet, werden zwei weitere Rosenbrock-Verfahren integriert. Das ROS3P-Verfahren ist ein dreistufiges Verfahren, welches aufgrund der zuvor genannten geschickten Wahl der Koeffizienten α_{ij} lediglich zwei Funktionsauswertungen benötigt. Das Verfahren ist A-stabil und besitzt die Konsistenzordnung drei [Lang & Verwer (2001)]. Die verwendeten Koeffizienten sind in Tabelle A.5 im Anhang enthalten. Das ROS4L-Verfahren aus [Hairer & Wanner (1996)] ist ein vierstufiges Verfahren der Konsistenzordnung vier, welches drei Funktionsauswertungen für die Durchführung eines Schrittes benötigt. Das Verfahren ist A-stabil und darüber hinaus ebenfalls L-stabil [Hairer & Wanner (1996)]. Die L-Stabilität spielt vor allem bei Systemen eine Rolle, bei denen auftretende Schwingungen in der Realität sehr schnell abklingen, d.h. bei Systemen mit Eigenwerten

mit betragsmäßig sehr großem Realteil. Die L-Stabilität sorgt durch eine numerische Dämpfung dafür, dass auch in der Simulation des Systems auftretende Oszillationen schnell gedämpft werden. Die verwendeten Parameter des ROS4L-Verfahrens sind in Tabelle A.6 im Anhang enthalten.

Die Rosenbrock-Wanner-Verfahren vereinen wesentliche Vorteile der Expliziten und Impliziten Runge-Kutta-Verfahren. Die zuvor vorgestellten Verfahren sind A-stabil und damit auch für die Simulation steifer Systeme geeignet. Dadurch, dass es sich um linear-implizite Verfahren handelt, ist der Aufwand pro Schritt fix, weswegen die Verfahren für Echtzeitanwendungen geeignet sind.

3.3.2.2 Behandlung von Time-Events

Bei Time-Events, also Unstetigkeiten, die nur von der Zeit abhängen, steht der Eintrittszeitpunkt schon zu Simulationsbeginn fest. Daher können diese auch bereits vor Simulationsbeginn behandelt werden, indem die Simulation in mehrere Teilsimulationen, die jeweils von einem Time-Event zum nächsten reichen, unterteilt wird. An den Zeitpunkten der Unstetigkeit wird das System mit den neuen Startbedingungen initialisiert und die Simulation fortgesetzt. Dazu wird in der Regel vor Simulationsbeginn die Simulation in die benötigten Teilsimulationen unterteilt. Aufgrund der unbestimmten Endzeit im Echtzeitbetrieb ist in diesem Fall jedoch eine komplette Vorausberechnung der Teilsimulationen nicht möglich. Weiterhin kann bei einer Echtzeitsimulation die Verwendung der Absolutzeit problematisch sein, da diese langfristig zu einem Überlaufen der Variablen führt.

Im Rahmen dieser Arbeit wird eine Methode entwickelt und implementiert, zeitabhängige Events in Echtzeitsimulationen ohne Verwendung der Absolutzeit zu behandeln. Dadurch ist sichergestellt, dass ein Überlaufen der Variablen der Simulationszeit nicht eintreten kann. An dieser Stelle wird erneut die Bedeutung von Modelica als Sprache zur Modellbildung deutlich. Innerhalb der Sprache stehen spezielle Sprachmittel zur Verfügung, die eine Definition von zyklisch auftretenden Events erlauben. Die Codegenerierung innerhalb des OpenModelica-Projektes wurde derart angepasst, dass entsprechender Code erzeugt wird, um diese Sprachmittel für Echtzeitsimulationen verfügbar zu machen.

Statt der Verwendung der Absolutzeit wird auf eine Relativzeit zurückgegriffen. Dazu wird vor Simulationsbeginn geprüft, wie häufig die einzelnen im Modell vorhandenen Time-Events eintreten. Unter Zuhilfenahme des Zyklustaktes der Steuerung wird ermittelt, wie viele Zyklen jeweils zwischen den Eintrittszeitpunkten zweier nachfolgender Events vergehen. Durch Verwendung eines Zählers für die vergangenen Zyklen seit dem letzten Eintreten des Events kann der nachfolgende Eintrittszeitpunkt des Events bestimmt werden. Die Simulation eines Schrittes erfolgt im Echtzeitbetrieb stets synchron zu der Zykluszeit der Steuerung. Da die Behandlung eines Time-Events eine Initialisierung des Systems mit den durch das Event geänderten Werten notwendig macht und somit den Beginn einer Teilsimulation

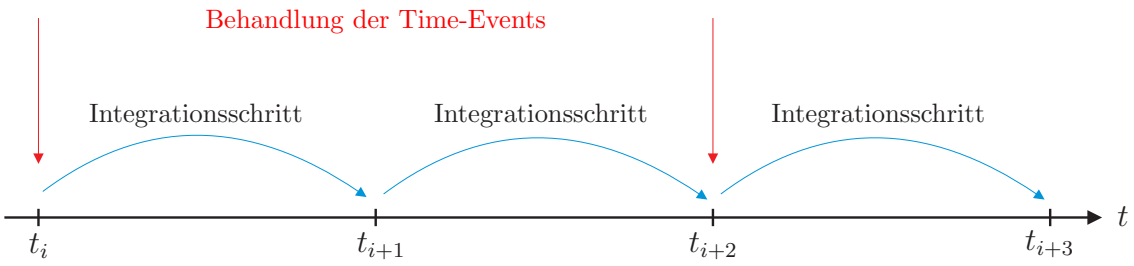

Abbildung 3.10: Behandlung von Time-Events am Beispiel eines Events, dessen Auftreten zyklisch in jedem zweiten Echtzeittakt erfolgt

bedeutet, ist es notwendig, dass der Zeitpunkt des Auftretens eines Time-Events ein Vielfaches der Zykluszeit ist. Auf diese Weise werden die Time-Events stets zu Beginn eines Echtzeitzyklus behandelt. Dies ist modellseitig zu beachten. Die Behandlung von Time-Events ist in Abbildung 3.10 graphisch veranschaulicht.

3.3.2.3 Behandlung von State-Events

Im Gegensatz zu Time-Events müssen die zustandsabhängigen State-Events während der Simulation behandelt werden, da der Zeitpunkt des Auftretens zu Simulationsbeginn nicht bekannt ist. Mithilfe einer Nullstellenfunktion wird in jedem Simulationsschritt geprüft, ob ein Event eingetreten ist. Die zu einem State-Event zugehörige Nullstellenfunktion ist eine eindimensionale, von den Zuständen abhängige Funktion, die bei Eintreten eines Events ihr Vorzeichen wechselt. Zu welchem exakten Zeitpunkt dieses Event aufgetreten ist, ist zunächst nicht bekannt. Der erste Schritt besteht darin, den Zeitpunkt des Eintretens zu bestimmen. Dies geschieht normalerweise iterativ durch eine Lokalisierung des Nulldurchgangs der zugrunde liegenden Nullstellenfunktion. Anschließend müssen die Zustände an diesem Zeitpunkt bestimmt werden. Diese können entweder durch eine Simulation, ausgehend vom letzten erfolgreichen Schritt, bis zu dem ermittelten Zeitpunkt des Events oder durch eine Interpolation bestimmt werden. Anschließend kann das Event behandelt werden.

Eine iterative Lösung zur Nullstellenbestimmung kann jedoch im Rahmen einer Echtzeitsimulation nicht verwendet werden. Es muss daher auf vereinfachte, nichtiterative Methoden zurückgegriffen werden. Zusätzlich ist, wie schon bei den Time-Events, darauf zu achten, dass anstelle der absoluten Zeit die relative Zeit seit dem letzten erfolgreichen Simulationsschritt verwendet wird. Die einfachste Möglichkeit ist es, auf eine genaue Lokalisierung der Nullstelle zu verzichten und das Event am Ende des Simulationsschrittes zu behandeln, d.h. das System mit den neuen Werten zu initialisieren. Anschließend wird die Simulation fortgesetzt. Diese Vorgehensweise benötigt, abgesehen von der Re-Initialisierung des Systems, keinen weiteren Aufwand für die Eventbehandlung, ist jedoch im Allgemeinen aufgrund der ungenügenden Genauigkeit nicht zu empfehlen.

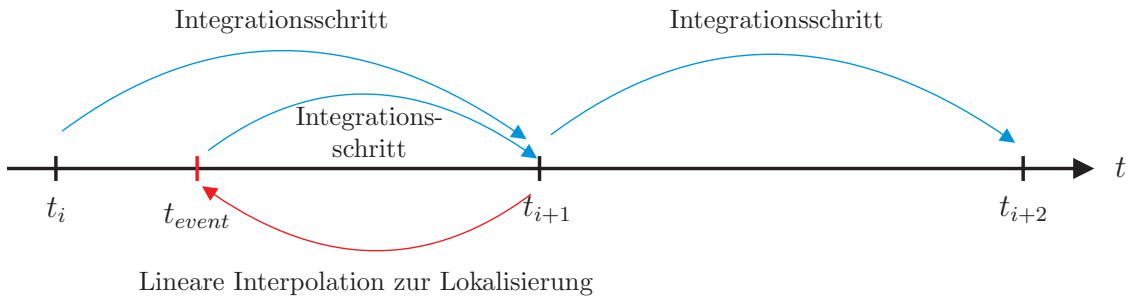

Abbildung 3.11: Behandlung von State-Events mit Herstellung der Synchronität zwischen Simulationszeit und Realzeit im aktuellen Schritt

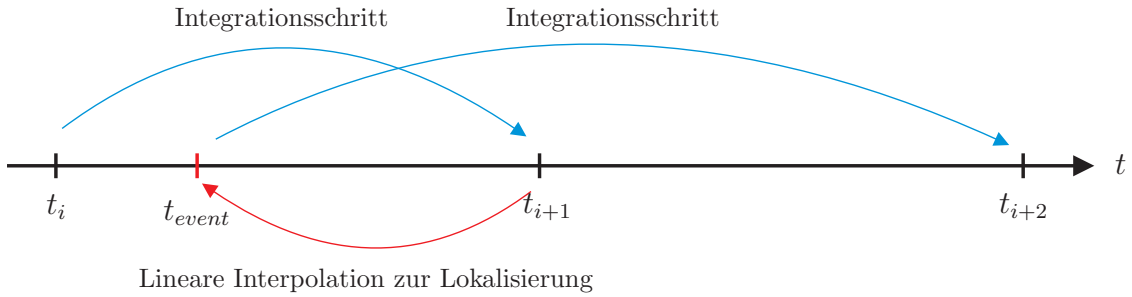

Abbildung 3.12: Behandlung von State-Events mit Herstellung der Synchronität zwischen Simulationszeit und Realzeit im nachfolgenden Schritt

Soll die Nullstelle lokalisiert werden, kann eine lineare Interpolation zur Lokalisierung verwendet werden [Kampfmann (2014)]. In diesem Fall wird zunächst mit Hilfe der linearen Interpolation der Zeitpunkt des Eintreffens des Events bestimmt. Die Zustände des Systems an dem ermittelten Eventzeitpunkt werden ebenfalls durch eine lineare Interpolation berechnet. Anschließend wird das Event behandelt. Es ist jedoch zu beachten, dass am Ende eines Simulationsschrittes die Synchronität von Simulationszeit und realer Zeit gewährleistet sein muss, was einen erneuten Integrationsschritt notwendig macht. Im Regelfall muss in einem Simulationsschritt daher ausreichend Zeit zur Verfügung stehen, um zunächst zu überprüfen, ob ein Event eingetreten ist, anschließend das Eintreten der Nullstelle zu lokalisieren, die Zustände zu diesem Zeitpunkt zu berechnen, die Unstetigkeit zu behandeln und anschließend einen weiteren Integrationsschritt durchzuführen, um die Synchronität zur realen Zeit wiederherzustellen. Diese Vorgehensweise ist in Abbildung 3.11 veranschaulicht.

Wird von einem Modell ausgegangen, bei dem nicht in jedem Simulationsschritt Events auftreten, kann das Herstellen der Synchronität auch in den nächsten Simulationsschritt verschoben werden, anstatt den zusätzlichen Simulationsschritt im aktuellen Schritt durchzuführen. Im folgenden Simulationsschritt ist ein entspre-

chend größerer Schritt zu simulieren [Kampfmann (2014)]. Dies ist in Abbildung 3.12 dargestellt.

Probleme bei der Eventlokalisierung mittels linearer Interpolation Abhängig von der Beschaffenheit der Nullstellenfunktion kann auch die Eventlokalisierung mittels linearer Interpolation zu Problemen führen. Dies ist der Fall, wenn ein zu früher Eintrittszeitpunkt des Events ermittelt wird, sodass im nächsten Simulationsschritt möglicherweise dasselbe Event erneut detektiert wird. Dieser Vorgang kann sich mehrfach wiederholen, sodass auch ein einzelnes Event zu einer nicht vorhersehbaren Laufzeit des Simulationsschrittes führen kann.

In der Praxis treten zusätzlich häufig Modelle mit mehreren Events pro Simulationsschritt auf. Diese sind für den Echtzeitbetrieb sehr problematisch, da zunächst jedes Event genau lokalisiert werden muss. In diesem Fall ist unter Umständen die lineare Interpolation nicht genau genug, sodass die korrekte Reihenfolge der Events nicht ermittelt werden kann. Zusätzlich ist das Einhalten der vorgegebenen Zykluszeit dadurch gefährdet, dass viele Teilsimulationen zwischen den einzelnen Events durchgeführt werden müssen. Ferner ist es möglich, dass das Eintreten eines Events bewirkt, dass ein anderes Event ausgelöst wird. In jedem Fall muss gewährleistet werden, dass sich das System nach Abschluss eines Simulationsschrittes in einem konsistenten Zustand befindet. Dies wird durch die *Event-Iteration* umgesetzt, bei der so lange alle auftretenden State-Events behandelt werden, bis dadurch keine weiteren Events ausgelöst werden. Da die Anzahl der zu behandelnden Events vorab nicht bekannt ist, ist die Verwendung von zustandsabhängigen Events im Echtzeitkontext stets kritisch. An dieser Stelle kann versucht werden, die maximale Anzahl an Iterationen zu begrenzen, allerdings ist in diesem Fall nicht gewährleistet, dass sich das System in einem konsistenten Zustand befindet.

Das Auftreten einer großen Anzahl an Events pro Simulationsschritt sollte bereits modellseitig vermieden werden. Dies soll im Folgenden anhand eines Beispiels verdeutlicht werden. Dazu wird das Verhalten eines Balles simuliert, der ausgehend von einer Ausgangshöhe von $s = 1\,\mathrm{m}$ auf einen Tisch fallen gelassen wird. Es ist offensichtlich, dass dieser aufgrund der Gravitationskraft zunächst beschleunigt wird und beim Auftreffen auf den Tisch von diesem abprallt und daraufhin seine Bewegungsrichtung umkehrt. Aufgrund von Reibung erreicht dieser seine ursprüngliche Höhe nicht mehr. Während zu Beginn eine relativ große Zeitspanne zwischen zwei aufeinanderfolgenden Kontakten zwischen Ball und Tisch vergehen, nimmt diese aufgrund der stetig geringer werdenden Flughöhe immer weiter ab, bis die Abstände so gering werden, dass die Bewegung des Balles vom Beobachter als Vibrieren auf dem Tisch wahrgenommen wird, bevor der Ball irgendwann schließlich vollständig zur Ruhe kommt. In der Simulation bedeutet dies, dass die auftretenden Events pro Simulationsschritt stetig zunehmen. Zu Beginn ist dies kein Problem, da lediglich in manchen Schritten ein Event auftritt, während des Zustandes des „Vibrierens“ tre-

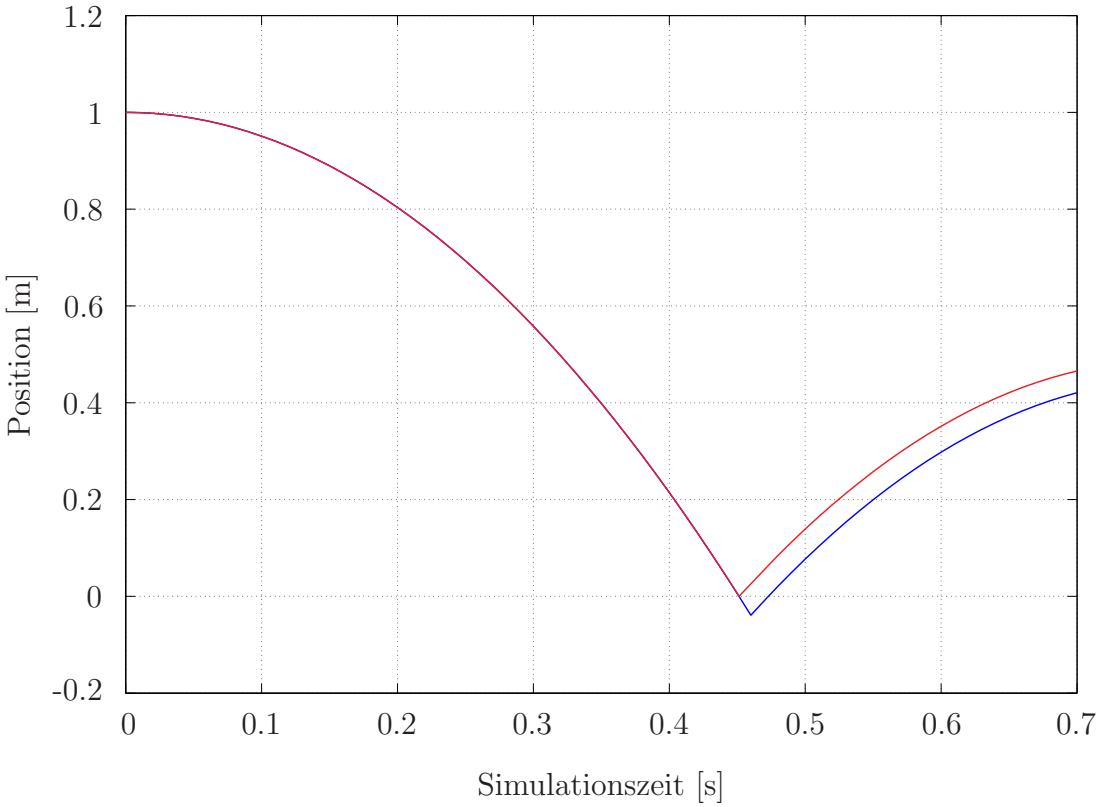

Abbildung 3.13: Simulation des Bouncing Ball Beispiels [Rot: mit linearer Interpolation zur Lokalisierung der Nullstelle; Blau: ohne Lokalisierung der Nullstelle] [Menager *et al.* (2015a)]

ten allerdings so viele Events auf, dass diese nicht mehr innerhalb des Echtzeittaktes behandelt werden können. Eine Echtzeitsimulation des Modells ist nicht möglich.

Wird das Modell jedoch derart angepasst, dass unterhalb einer gewissen Geschwindigkeit des Balles mit der Geschwindigkeit von Null initialisiert wird, also das Vibrieren modellseitig unterdrückt wird, kann die Vielzahl der Events verhindert werden. Dies sorgt dafür, dass das Modell im Echtzeittakt gerechnet werden kann. Der auftretende Informationsverlust ist dabei so gering, dass er in der Regel vernachlässigt werden kann [Kampfmann (2014)]. Ein derartiges Verfahren ist jedoch in keinem Fall allgemeingültig, sondern stark modellabhängig. Insbesondere ist auch in diesem Fall die Anzahl der maximalen Events im Vorfeld nicht vorhersehbar, sondern muss durch Laufzeittests ermittelt werden. Abbildung 3.13 zeigt die Simulation des *Bouncing Ball* Beispiels graphisch. Der blaue Graph zeigt das Simulationsergebnis, wenn auf eine Lokalisierung der Nullstelle verzichtet wird. Das Event wird am Ende eines Simulationsschrittes erkannt und behandelt. Zu diesem Zeitpunkt befindet sich der Ball jedoch bereits 39 mm unterhalb des Tisches. Bei Verwendung einer linearen Interpolation zur Lokalisierung (roter Graph) kann die Genauigkeit deut-

lich gesteigert werden. Der Ball durchdringt den Tisch in diesem Fall lediglich um 10^{-9} mm. Die Güte der linearen Interpolation hängt im Wesentlichen von dem Verlauf der Nullstellenfunktion ab. Ist dieser annähernd linear, können mit der linearen Interpolation sehr gute Ergebnisse erzielt werden.

3.3.2.4 Behandlung von algebraischen Schleifen

Algebraische Schleifen im Modell sind, wie schon State-Events, grundsätzlich für den Echtzeitbetrieb nicht zulässig. Diese müssen iterativ gelöst werden, beispielsweise mit dem Newton-Verfahren (vgl. Abschnitt 2.1.3.2). Für iterative Löser ist es nicht möglich, eine obere Grenze der Laufzeit anzugeben, weswegen die Einhaltung des Echtzeittaktes nicht gewährleistet werden kann. Handelt es sich bei den Schleifen um lineare algebraische Schleifen, können diese jedoch toleriert werden, da in diesem Fall eine Iteration des Newton-Verfahrens ausreicht, um die Lösung zu erlangen[1].

Es ist zwar möglich, mithilfe von guten Startschätzungen und symbolischen Jacobi-Matrizen auch für nichtlineare algebraische Schleifen eine schnelle Konvergenz zu ermöglichen, eine Abschätzung für die Laufzeit ist dennoch nicht möglich. Abhilfe kann es schaffen, wenn die Anzahl der durchgeführten Iterationen begrenzt wird. In diesem Fall ist jedoch nicht gewährleistet, dass eine ausreichend genaue Lösung gefunden wird. Derartige Techniken können modellspezifisch verwendet werden, sind aber keinesfalls allgemeingültig. Insgesamt ist die Forderung, komplett auf algebraische Schleifen zu verzichten, eine sehr große Einschränkung für die verwendbaren Modelle. In Abhängigkeit von dem Anwendungsfall kann es möglich sein, Modelle mit algebraischen Schleifen dennoch zu verwenden. Dies kann beispielsweise der Fall sein, wenn das Modell nicht zur Erfüllung von Regelungsaufgaben auf der Steuerung ausgeführt wird, sondern lediglich parallel zur Steuerungsapplikation beispielsweise zu Diagnosezwecken simuliert wird. Des Weiteren kann es vorkommen, dass die Lösung $\mathbf{x}_n$ der algebraischen Schleife nicht von dem vorherigen Wert $\mathbf{x}_{n-1}$ abhängt. In diesem Fall wird in jedem Simulationsschritt ein unabhängiges Gleichungssystem gelöst. Wird das Modell beispielsweise lediglich zur Diagnose oder für eine Vorsteuerung verwendet, ist es nicht zwingend notwendig, in jedem Schritt eine aktuelle Lösung zu erhalten. Kann in einem Zeitschritt eine Lösung nicht ermittelt werden, wird stattdessen versucht, das Gleichungssystem im nächsten Schritt mit den neuen Werten zu lösen. Dieses Vorgehen funktioniert jedoch nicht, sofern die Lösung $\mathbf{x}_n$ von der vorherigen Lösung $\mathbf{x}_{n-1}$ abhängt. Allgemein gilt, dass immer dann, wenn die Einhaltung des Echtzeittaktes nicht zwingend erforderlich ist und eine Zykluszeitüberschreitung toleriert werden kann, eine Verwendung von Modellen mit algebraischen Schleifen unter Umständen möglich ist.

[1] Dies gilt zumindest, sofern die Jacobi-Matrix in symbolischer Form vorliegt. Wird die Jacobi-Matrix numerisch bestimmt, können auch bei linearen Gleichungssystemen, bedingt durch die numerischen Verfahren und damit einhergehenden Ungenauigkeiten, mehrere Iterationen notwendig sein.

3.3.2.5 Zusammenfassung und Fazit

Zusammenfassend lässt sich sagen, dass die Voraussetzung der Echtzeitfähigkeit einer Simulation eine deutliche Einschränkung ist. In erster Linie betreffen die Einschränkungen die Modelle, sodass die Anforderung der Echtzeitfähigkeit bereits bei der Modellierung berücksichtigt werden muss. Abhängig vom Einsatzgebiet des Modells ist unter anderem strikt darauf zu achten, dass das Modell keine algebraischen Schleifen enthält. Liegen ausschließlich lineare algebraische Schleifen vor, kann dies unter Umständen toleriert werden. Weiterhin ist es notwendig, die Zahl der auftretenden Events so gering wie möglich zu halten. Zeitabhängige Events sind in der Regel unproblematischer als zustandsabhängige Events. Es ist jedoch darauf zu achten, dass lediglich die relative Zeit verwendet wird, da die absolute Zeit im Echtzeitbetrieb langfristig zu einem Überlauf der Variablen führt. Dazu ist im Rahmen dieser Arbeit die Codegenerierung derart angepasst worden, dass die Verwendung von Time-Events, solange der Eintrittszeitpunkt stets ein Vielfaches der Zykluszeit der Steuerung ist, problemlos möglich ist. Die Verwendung von zustandsabhängigen Events ist deutlich komplizierter. Für die Lokalisierung des exakten Eintreffens des Events werden iterative Methoden benötigt. Sofern die Anzahl der auftretenden State-Events gering ist, kann versucht werden, den Nulldurchgang der zugrunde liegenden Nullstellenfunktion mit Näherungsmethoden wie beispielsweise einer linearen Interpolation zu bestimmen. Diese Methode funktioniert selbstverständlich bei Nullstellenfunktionen, die linear oder annähernd linear sind, deutlich besser als bei stark nichtlinearen Nullstellenfunktionen.

Grundsätzlich können die zuvor beschriebenen Maßnahmen als Einschränkungen für die Modelle aufgefasst werden, was bedeutet, dass unter Umständen Anpassungen an bereits vorhandenen Modellen der Auslegungsphase notwendig werden, um diese in Echtzeit auf der Steuerungshardware zu verwenden. Diese Anpassungen widersprechen auf den ersten Blick dem Prinzip der durchgängigen Verwendung der Modelle. Durch die Prinzipien der objektorientierten Modellierung kann das Problem jedoch elegant gelöst werden, indem mehrere Modelle der einzelnen Komponenten mit unterschiedlichem Detaillierungsgrad vorhanden sind. Es ist beispielsweise möglich, das Modell eines hydraulischen Zylinders unter Verwendung eines komplexen Reibungsmodells sowie komplexen Modellen der Anschläge zu erstellen. Ein derartiges Modell ist in der Regel nicht echtzeitfähig. Gleichzeitig kann ein alternatives Modell des Zylinders mit einem einfachen Reibungsmodell und vereinfachten Anschlägen aufgebaut werden, welches unter Umständen für eine Echtzeitsimulation geeignet ist. Bei einer Überführung eines Modells von der Auslegungs- in die Inbetriebnahmephase sind lediglich die entsprechenden nicht-echtzeitfähigen Komponenten durch die echtzeitfähigen Komponenten auszutauschen. Durch die Verwendung spezieller Modelica-Sprachmittel kann der Komponentenaustausch auf einfache Weise, in der Regel sogar graphisch durch Umschalten der Komponente im Parameterfenster, durchgeführt werden.

Es besteht darüber hinaus, sofern eine Anpassung des Modells nicht erwünscht ist, natürlich die Möglichkeit, für ein spezielles Modell zu testen, ob die Simulation auf der Echtzeithardware in Echtzeit durchgeführt werden kann, auch wenn dieses Modell nichtlineare algebraische Schleifen oder State-Events enthält. Diese Modelle können aber nur unter speziellen Bedingungen auf der Hardware ausgeführt werden und sind für den Betrieb in einer echtzeitkritischen Anwendung nicht zugelassen, da keine Garantie gegeben werden kann, ob es nicht zu einem späteren Zeitpunkt, möglicherweise bei einer ungünstigen Parameterkombination, zu einer Überschreitung des Echtzeittaktes kommt. Für die Durchführung einer virtuellen Inbetriebnahme, bei der keine harten Echtzeitforderungen gelten, kann dies beispielsweise jedoch zulässig sein.

Neben dem Modell selbst spielt die Wahl des numerischen Verfahrens zum Lösen der Modellgleichungen eine große Rolle für die Echtzeitfähigkeit. Wie bereits zuvor begründet, ist ein echtzeitfähiges numerisches Lösungsverfahren ein Verfahren mit fester Schrittweite und ohne iterative Komponenten. Dadurch unterscheiden sich diese Verfahren deutlich von den klassischerweise innerhalb von Offline-Simulationen eingesetzten Verfahren. Abhängig von dem gewählten Verfahren wird eine unterschiedliche Anzahl an Funktionsauswertungen innerhalb eines Taktes benötigt. Für einfache Systeme, wie sie häufig bei Reglermodellen auftreten, kann es beispielsweise ausreichend sein, das Explizite Euler-Verfahren zur Lösung zu verwenden, welches lediglich eine Funktionsauswertung benötigt, während Streckenmodelle hydraulischer Systeme häufig mathematisch steif sind, was robustere Verfahren wie die Rosenbrock-Verfahren notwendig macht. Abhängig von der Ordnung des verwendeten Rosenbrock-Verfahrens werden dazu jedoch stets mehrere Funktionsauswertungen benötigt. Grundsätzlich muss für jedes Modell das numerische Verfahren sowie der verwendete Echtzeittakt sinnvoll gewählt werden.

3.3.3 Integration des Codes in die Steuerungshardware

Der letzte Schritt bei der Realisierung einer Werkzeugkette zur Ausführung von Modellen auf Steuerungshardware besteht darin, den aus dem Modelica-Modell generierten C++-Code gemeinsam mit dem echtzeitfähigen Simulationskern auf dem Zielsystem auszuführen. Entscheidend ist in diesem Zusammenhang die Integration der Bibliotheken in die Laufzeitumgebung der Steuerungshardware. In bisherigen Entwicklungsprozessen ist der Schritt der Integration des Codes häufig problematisch, da zur Umsetzung entsprechende Schnittstellen der Steuerung benötigt werden, die für Industriesteuerungen in vielen Fällen nicht zur Verfügung stehen. Aus diesem Grund werden bisher, sofern modellbasierte Entwicklungsprozesse angewendet werden, zumeist spezielle Prototyping-Plattformen verwendet, die die Ausführung von Code auf einfache Weise unterstützen. Diese erlauben zwar ein schnelles Testen des Reglercodes, aufgrund der Tatsache, dass diese speziellen Entwicklungsplattformen später im Betrieb jedoch nicht weiterverwendet werden, bleibt das Problem der Re-Implementierung der Steuerungsapplikation auf der Industrie-

steuerung bestehen. Aus diesem Grund ist es wünschenswert, den generierten Code unmittelbar auf der letztendlich verwendeten Steuerungshardware auszuführen.

Grundsätzlich bestehen mehrere Möglichkeiten, den Code in die Laufzeitumgebung der Steuerung zu integrieren. Die erste Möglichkeit besteht darin, die Steuerungshardware als reine Echtzeithardware zu betrachten und den Code unmittelbar, d.h. entkoppelt von einer Steuerungsapplikation, auf dieser auszuführen. Diese Vorgehensweise hat den Nachteil, dass der Applikationsingenieur, der für die Erstellung der Steuerungsapplikation zuständig ist, die ihm gewohnte Entwicklungsumgebung nicht weiter verwenden kann und sich daher in neue Werkzeuge einarbeiten muss. Auf der anderen Seite bietet dieses Vorgehen dadurch jedoch eine hohe Flexibilität bezüglich der Einbindung der Hardware in beliebige andere Entwicklungsumgebungen. Die zweite Möglichkeit besteht darin, den Modellcode nicht unabhängig von einer Steuerungsapplikation zu verwenden, sondern diesen in eine bestehende Applikation innerhalb der standardmäßig verwendeten Entwicklungsumgebung zu integrieren. Dies hat den Vorteil, dass eine graphische Oberfläche zur Verfügung steht und der Ingenieur weiterhin in seiner gewohnten Umgebung arbeiten kann. Die Integration erfolgt in diesem Fall beispielsweise mit Hilfe eines klassischen SPS-Funktionsbausteins, in dem der Code gekapselt wird.

Im Rahmen dieser Arbeit werden, wie bereits zuvor erwähnt, entweder IndraControl XM22 Industriesteuerungen oder IndraControl VPB40.3 Industrie-PCs eingesetzt, auf denen das Echtzeitbetriebssystem VxWorks läuft. Für eine Integration des Codes in die Steuerungshardware ist es, unabhängig welche der beiden zuvor vorgestellten Varianten verwendet wird, in jedem Fall zunächst notwendig, den C++-Code für das Betriebssystem VxWorks zu kompilieren. Dazu wird ein VxWorks-Compiler[1] der Firma Windriver verwendet. Das Ergebnis des Kompiliervorgangs sind zwei Bibliotheken, von denen eine den Simulationskern und die andere das Modell enthält. Es ist zu beachten, dass der Simulationskern lediglich einmalig kompiliert werden muss und schließlich unverändert für sämtliche Modelle verwendet werden kann.

Ein Beispiel für die zuvor beschriebene Schnittstelle, wie sie für eine erfolgreiche Integration des Codes in die Steuerungshardware benötigt wird, stellt das *Motion Logic Programming Interface* von Bosch Rexroth dar, dessen Funktionsweise nachfolgend in Abschnitt 3.3.3.1 beschrieben wird. Sowohl die zuvor genannte Industriesteuerung als auch der genannte Industrie-PC werden von der Schnittstelle unterstützt. Die beiden Methoden der Integration des Codes in die Laufzeitumgebung der Steuerung werden in den Abschnitten 3.3.3.2 (Integration ohne Funktionsblock) sowie 3.3.3.3 (Integration mit Funktionsblock) behandelt.

3.3.3.1 Motion Logic Programming Interface

Das Motion Logic Programming Interface (MLPI) ist eine offene Schnittstelle für Industriesteuerungen von Bosch Rexroth [Engels & Gabler (2012)]. Das Ziel der

[1] `http://www.windriver.com/products/development-tools/`

Schnittstelle ist es, sämtliche Steuerungsfunktionen von außen unter Verwendung von Hochsprachen wie C/C++ oder Java zugänglich zu machen. Weiterhin existieren Implementierungen der Schnittstelle für unterschiedliche Softwareframeworks wie MATLAB/Simulink (mlpi4MATLAB) oder LabVIEW (mlpi4LabVIEW). Auch in Form einer Modelica-Bibliothek (mlpi4Modelica) steht die Schnittstelle zur Verfügung.

Das Motion Logic Programming Interface ist modular aufgebaut und besteht intern aus mehreren Bibliotheken, die jeweils Zugriff auf spezielle Funktionen der Steuerung ermöglichen.

- **mlpiApiLib**: Diese Bibliothek ist hauptsächlich für den Aufbau einer Verbindung zu einer Steuerung und deren Beendigung zuständig.
- **mlpiSystemLib**: Diese Bibliothek stellt Methoden bereit, um den Systemzustand der Hardware auszulesen, beispielsweise die aktuelle Temperatur der Steuerung oder die verwendete Firmwareversion. Weiterhin ist ein Zugriff auf Diagnosedaten im Fall eines aufgetretenen Fehlers möglich.
- **mlpiParameterLib**: Diese Bibliothek erlaubt den Zugriff auf die Steuerungs- sowie Antriebsparameter. Es stehen Funktionen sowohl zum Schreiben, sofern der Parameter nicht schreibgeschützt ist, als auch zum Lesen bereit.
- **mlpiMotionLib**: Diese Bibliothek erlaubt die Verwendung von Motion-Befehlen, um Antriebe zu bewegen. Dazu stehen unter anderem die Befehle der *PLCopen*-Bibliothek in Form von Funktionen bereit. PLCopen ist ein Standard für Positionieraufgaben für den Einachs- sowie Mehrachsbetrieb. Es sind beispielsweise Funktionen zum Ansteuern von vorgegebenen Positionen (MC_MoveAbsolute) oder zur Vorgabe einer Geschwindigkeit für eine Achse (MC_MoveVelocity) vorhanden[1].
- **mlpiTaskLib**: Diese Bibliothek erlaubt die Konfiguration von Tasks. Weiterhin werden Funktionen bereitgestellt, die eine extern getriggerte Abarbeitung der Tasks auf der Steuerung ermöglichen[2].
- **mlpiTraceLib**: Diese Bibliothek kann verwendet werden, um Debug-Ausgaben zu verarbeiten. Dafür werden spezielle Ausgaben in den ausgeführten Code integriert, die anschließend für eine Auswertung ausgelesen werden können.
- **mlpiWatchdogLib**: Diese Bibliothek wird verwendet, um die Applikation zu überwachen. Dafür stehen Methoden zum Starten, Stoppen oder Zurücksetzen

[1] Wie die Funktionen unmittelbar zur modellbasierten Entwicklung von Regelungsalgorithmen verwendet werden können, wird in Abschnitt 4.2 am Beispiel einer komplexen Flaschenabfüllanlage gezeigt.

[2] Auf diese Funktionalität wird im weiteren Verlauf der Arbeit in Abschnitt 3.4.2 noch näher eingegangen.

des Watchdogs bereit. Eine Möglichkeit ist die Überwachung der Überschreitung des Echtzeitzyklus.

- **mlpiLogicLib**: Diese Bibliothek beinhaltet Funktionen, um die Applikation der Steuerung zu starten, zu stoppen oder neuzustarten. Weiterhin werden Funktionen bereitgestellt, um auf in der Steuerungsapplikation vorhandene Symbolvariablen zuzugreifen und diese Werte zu lesen oder zu schreiben. Die im Rahmen dieses Kapitels benötigten Methoden zur Integration von eigenem Code in die Steuerungsapplikation befinden sich ebenfalls in dieser Bibliothek.

- **mlpiContainerLib**: Diese Bibliothek beinhaltet Methoden, um große Datenmengen zyklisch zwischen Steuerung und externer Applikation zu transferieren. Dazu wird nicht jeder Wert einzeln übertragen, sondern ein Container verwendet, der im Gesamten übertragen wird.

- **mlpiIoLib**: Diese Bibliothek beinhaltet Methoden, um auf angeschlossene Feldbus-I/Os zuzugreifen, um Werte lesen sowie schreiben zu können.

3.3.3.2 Integration ohne Funktionsblock

Die erste Möglichkeit, den generierten Code in die Laufzeitumgebung der Steuerung einzubinden, besteht darin, den Code losgelöst von einer Steuerungsapplikation auf der Hardware auszuführen. In diesem Fall kann die Steuerung als Echtzeitrechner betrachtet werden, der es ermöglicht, beliebigen Code auszuführen. Der auszuführende Code wird kompiliert auf der Steuerung abgelegt und wird unmittelbar beim Hochfahren der Steuerung geladen und ausgeführt. Dies entspricht grundsätzlich dem heute üblichen Vorgehen bei Verwendung einer speziellen Prototyping-Plattform. Es ist jedoch zu beachten, dass in diesem Fall bereits die später an der realen Anlage verwendete Hardware verwendet wird. Steht darüber hinaus ein Interface zur Steuerungsfirmware zur Verfügung, kann auf die Steuerungsfeatures zugegriffen werden, indem die entsprechenden Funktionen zum Steuerungszugriff in dem ausgeführten Code verwendet werden. Die zuvor in Abschnitt 3.3.3.1 vorgestellte, offene Schnittstelle des Motion Logic Programming Interface stellt beispielsweise derartige Funktionen zur Verfügung.

Da bei dieser Variante der Codeintegration nicht zwingend eine klassische Steuerungsapplikation auf der Steuerung vorhanden ist, in der üblicherweise die einzelnen Tasks zum Aufruf der Programme der Steuerungsapplikation verwaltet und konfiguriert werden, muss das Anlegen einer Task zum zyklischen Aufruf des Codes eigenständig vorgenommen werden. Auch die Konfiguration der angeschlossenen I/O-Module sowie Antriebe müssen eigenhändig und ohne Unterstützung durch eine graphische Benutzeroberfläche innerhalb des Steuerungscodes konfiguriert werden. Die dazu benötigten Funktionen sind ebenfalls Teil des Motion Logic Programming Interface, eine Verwendung einer speziellen Entwicklungsumgebung ist in diesem Fall nicht erforderlich. Dadurch ist es möglich, die Steuerungshardware in beliebige,

eigene Entwicklungsumgebungen zu integrieren, wodurch sich ein hoher Grad an Flexibilität ergibt.

3.3.3.3 Integration mit Funktionsblock

Neben der zuvor vorgestellten Möglichkeit ist es möglich, den Code mit Hilfe eines SPS-Funktionsblocks in die Laufzeitumgebung der Steuerung zu integrieren. Diese Vorgehensweise ermöglicht es, den Code mit einer bestehenden Steuerungsapplikation zu verknüpfen. Der Funktionsblock dient dabei als Schnittstelle zwischen der Steuerungsapplikation und der Bibliothek, die den Modellcode und den Simulationskern enthält und an den Funktionsblock angehängt werden soll. Damit der Code in einen Funktionsblock integriert werden kann, werden entsprechende Schnittstellenfunktionen der Steuerung benötigt. Derartige Funktionen stellt das zuvor beschriebenen Motion Logic Programming Interface zur Verfügung. Jeder Funktionsblock in einer SPS-Applikation besteht aus einem Deklarationsteil, in dem die Variablen für den Funktionsblock angelegt werden, und einem Implementierungsteil, der die Implementierung enthält. Die Implementierung eines Funktionsblocks liegt standardmäßig in einer der SPS-Programmiersprachen nach IEC 61131-3 vor. Durch die Integration des zuvor generierten Codes in den Funktionsblock entfällt diese Implementierung in SPS-Programmiersprachen, stattdessen wird der generierte Code verwendet. Die innerhalb des Deklarationsteils definierten Variablen des Funktionsblocks ermöglichen einen Austausch zwischen den Variablen des Modells im generierten Code und anderen Teilen der SPS-Applikation. Aus diesem Grund ist es notwendig, dass sämtliche Modellvariablen, die innerhalb der Applikation verwendet werden sollen, innerhalb des Deklarationsteils des Funktionsblocks angegeben werden.

Ferner ist es notwendig, für jeden Funktionsblock eine Initialisierungsmethode sowie eine Methode, die zyklisch aufgerufen wird, zu implementieren. Innerhalb der Initialisierungsmethode, die von der Steuerungsfirmware bereits beim Laden der Applikation aufgerufen wird, wird der Simulationskern initialisiert und das auszuführende Modell geladen. Die zyklisch aufgerufene Methode enthält lediglich den Aufruf des numerischen Verfahrens zur Durchführung eines Simulationsschrittes. Zusätzlich werden zu Beginn eines jeden Simulationsschrittes die Eingangsvariablen aus dem Funktionsblock in das Modell geschrieben und nach Beendigung des Schrittes die Ausgangsgrößen des Modells in die Ausgänge des Funktionsblocks geschrieben. Die Länge eines Simulationsschrittes entspricht der eingestellten Zykluszeit der Steuerung, um eine synchrone Ausführung in Echtzeit zu gewährleisten.

Bei Verwendung der Integration des Codes mit Hilfe eines Funktionsblocks besteht der Vorteil, dass der Applikationsingenieur weiterhin in seiner gewohnten Umgebung arbeiten kann. Dieser Funktionsblock kann wie jeder andere SPS-Funktionsblock verwendet werden. Der einzige Unterschied ist, dass dieser keine Implementierung in SPS-Programmiersprachen enthält. Stattdessen wird die zuvor automatisiert generierte Bibliothek zu dem Baustein hinzugelinkt. Für eine möglichst automatisierte,

einfache Integration des Codes wird der Funktionsblock bei der Codegenerierung aus der Modellbeschreibung ebenfalls automatisch generiert. Die im Modell vorhandenen Variablen werden dabei automatisch in den Deklarationsteil des Funktionsblocks eingetragen. Die Generierung des Funktionsblocks erfolgt unter Verwendung des *PLCopen XML Standards* [Esteez *et al.* (2009)]. Für die Integration des Codes in die Steuerungsapplikation ist daher lediglich der generierte Funktionsblock in die Applikation zu importieren. Innerhalb einer Applikation können Funktionsblöcke, die eine Implementierung in klassischen SPS-Sprachen besitzen, mit solchen, deren Implementierung in C++ vorliegt, kombiniert werden. Weiterhin ist es möglich, verschiedene Funktionsblöcke mit unterschiedlichen Implementierungen in einer Applikation miteinander zu verbinden.

3.3.3.4 Zusammenfassung und Fazit

Im Rahmen dieser Arbeit besteht das Ziel, einen durchgängigen, modellbasierten Entwicklungsprozess zu entwickeln, der in der Praxis möglichst einfach eingesetzt werden kann. Aus diesem Grund ist darauf zu achten, dass die Applikationserstellung möglichst auf die gleiche Art erfolgt wie bisher. Diese erfolgt bei Verwendung der in dieser Arbeit verwendeten IndraControl XM22 Industriesteuerungen bzw. der IndraControl VPB40.3 Industrie-PCs in der Regel mit dem von Bosch Rexroth entwickelten Tool *IndraWorks Engineering*[1]. Das Tool erlaubt die Erstellung einer Steuerungsapplikation sowie das Hinzufügen der verwendeten Peripheriegeräte (I/O-Module sowie Antriebe). Des Weiteren stehen innerhalb von IndraWorks umfangreiche Werkzeuge zur Diagnose der Applikation sowie der Visualisierung der Variablen bereit. Es wird daher im weiteren Verlauf dieser Arbeit die Methode der Integration über einen Funktionsblock gewählt. Diese erlaubt die komfortable Einbindung von Funktionsblöcken, deren Implementierung in C/C++ vorliegt, in bestehende Applikation sowie die Verschaltung mehrerer Funktionsblöcke mit unterschiedlichen Implementierungen miteinander.

3.4 Virtuelle Inbetriebnahme zur Validierung der Steuerungsapplikation

Im vorherigen Abschnitt wurde ausführlich beschrieben, wie im Sinne einer Durchgängigkeit des Entwicklungsprozesses bereits modelliertes Wissen aus der Auslegungs- in die Inbetriebnahmephase überführt werden kann. Die Verwendung der vorgestellten Werkzeugkette erlaubt es, aus Simulationsmodellen in der offenen Modellierungssprache Modelica ausführbaren Code zu erzeugen und diesen automatisiert auf die verwendete Industriesteuerung zu übertragen und dort auszuführen. Im nächsten Schritt des Produktentwicklungsprozesses (vgl. Abbildung 3.3) folgt nun die Validierung der Steuerungsapplikation. Es ist zu beachten, dass diese in

[1] `www.boschrexroth.com/indraworks`

der Regel nicht nur den aus der Simulationsumgebung generierten Code, sondern eine Vielzahl weiterer Funktionen, beispielsweise in der Firmware standardmäßig verfügbare Safety-Bausteine, enthält. Durch die Verwendung modellbasierter Entwicklungsmethoden kann der Test der Applikation bereits erfolgen, bevor die reale Anlage existiert. Die Erprobung, Optimierung und Parametrierung der zum Betrieb erforderlichen Software an der Anlage stellt eine wesentliche Aufgabe innerhalb der Phase der Inbetriebnahme dar. Wird die Erprobung und Optimierung des Reglercodes auf der Steuerungshardware mit Hilfe eines virtuellen Abbildes des Systems durchgeführt, wird von einer *virtuellen Inbetriebnahme* gesprochen.

Durch die Methoden der modellbasierten Entwicklung, insbesondere die Codegenerierung sowie die virtuellen Inbetriebnahme, gelingt es, dass die Phasen der Auslegung und Inbetriebnahme miteinander verschmelzen. Da für den Beginn der Inbetriebnahme in diesem Fall nicht auf die Fertigung und Montage der Anlage gewartet werden muss, können beide Phasen ohne zeitlichen Verzug unmittelbar gemeinsam erfolgen, wodurch sich eine deutliche Effizienzsteigerung innerhalb des Entwicklungsprozesses ergibt. An der realen Anlage ist schließlich lediglich eine Feinoptimierung der Parameter in einer bereits validierten Software durchzuführen. Für die Erprobung und Optimierung des Reglercodes stellt die Simulationsumgebung, in der bereits in der Auslegungsphase ein virtuelles Abbild des betrachteten Systems in Form eines Simulationsmodells erzeugt wird, eine geeignete Umgebung dar. Eine Möglichkeit, den Funktionstest mit Hilfe einer virtuellen Anlage durchzuführen, ist die *Hardware-In-The-Loop-Simulation* (HiL-Simulation). Die HiL-Simulation integriert reale Hardware in den Simulationsprozess, sodass Teile der Simulation rein virtuell als Modell vorliegen, während andere Komponenten real vorhanden sind. Weiterhin besteht auch die Möglichkeit, den Funktionstest mit Hilfe einer *Software-In-The-Loop-Simulation* (SiL-Simulation) rein virtuell durchzuführen. In diesem Fall ist keine reale Hardware notwendig, stattdessen wird die Steuerungsfirmware auf einem herkömmlichen Computer, auf dem auch die Simulation der Regelstrecke durchgeführt wird, ausgeführt. Anschließend kann die virtuelle Steuerung wie eine reale Steuerung verwendet werden. In diesem Fall wird auch von einer *Soft-SPS* gesprochen [Seitz (2015)]. Es ist zu beachten, dass bei Verwendung einer Soft-SPS im Kontext einer SiL-Simulation die Ausführung des Codes in harter Echtzeit nicht garantiert werden kann. Im Folgenden wird stets die virtuelle Inbetriebnahme mit Hilfe der HiL-Simulation betrachtet.

Die einfachste Möglichkeit für die Durchführung der virtuellen Inbetriebnahme stellt die Simulation der Regelstrecke in Echtzeit dar. In diesem Fall sind die beiden Komponenten, die virtuelle Regelstrecke sowie die Steuerungshardware, automatisch stets synchron. Für die Durchführung der Simulation in Echtzeit ist eine spezielle Echtzeithardware notwendig. Eine solche Echtzeithardware steht jedoch mit der verwendeten Steuerungshardware bereit. Die in Abschnitt 3.3 vorgestellte Werkzeugkette kann dazu verwendet werden, sowohl aus dem Modell des Reglers als auch aus dem Modell der Regelstrecke Code zu generieren und diesen auf der

Steuerungshardware auszuführen. Die Methode der Erprobung und Optimierung des automatisiert generierten Codes auf der Steuerungshardware mit Hilfe einer virtuellen Inbetriebnahme in Echtzeit wird in Abschnitt 3.4.1 vorgestellt.

Sollen komplexe Regelstreckenmodelle verwendet werden, deren Simulation in Echtzeit nicht möglich ist, kann die zuvor genannte Methode nicht verwendet werden. In diesem Fall tritt bei einer HiL-Simulation das Problem auf, dass die an der Simulation beteiligten Komponenten mit unterschiedlichen Geschwindigkeiten ablaufen, da die Steuerungsapplikation weiterhin in Echtzeit ausgeführt wird. Da die Wahrung der Synchronität zwischen Simulation und Steuerungshardware zur Erlangung von aussagekräftigen Simulationsergebnissen essentiell ist, sind an dieser Stelle spezielle Verfahren notwendig, die die Synchronität dennoch ermöglichen. Nach derzeitigem Stand der Technik steht keine Möglichkeit bereit, eine echtzeitfähige Steuerungshardware mit einer nicht-echtzeitfähigen Simulationsumgebung derart zu koppeln, dass diese synchronisiert sind. Im Rahmen dieser Arbeit wird in Abschnitt 3.4.2 ein Verfahren entwickelt, welches eine synchronisierte HiL-Simulation zwischen Simulationsumgebung und Steuerungshardware auch dann ermöglicht, wenn die Simulation der Regelstrecke nicht in Echtzeit abläuft.

3.4.1 Virtuelle Inbetriebnahme unter Echtzeitbedingungen

Die Durchführung einer virtuellen Inbetriebnahme unter Echtzeitbedingungen stellt bisher den Regelfall einer virtuellen Inbetriebnahme dar. In diesem Fall wird zur Simulation der Regelstrecke eine Echtzeithardware benötigt. Als Hardwareplattform zur Ausführung des Modells kann jedoch die ohnehin vorhandene Steuerungshardware verwendet werden, die sich, wie bereits im letzten Abschnitt erläutert, als Echtzeitrechner auffassen lässt. Gelingt es, diese Hardware sowohl für die eigentliche Regelungsaufgabe als auch für die Simulation der Regelstrecke zu verwenden, kann eine zusätzliche Echtzeithardware entfallen.

Es stehen bereits einige Softwarewerkzeuge bereit, die eine virtuelle Inbetriebnahme unter Echtzeitbedingungen erlauben. Zwei Beispiele dafür stellen die Tools *ISG-virtuos*[1] der Firma ISG sowie *industrialPhysics*[2] der Firma *machineering GmbH & Co. KG* dar. Diese Tools nutzen die angeschlossene Steuerungshardware als Echtzeitrechner und erzeugen aus dem Simulationsmodell ausführbaren Code, den sie auf der Hardware ausführen. Bei Verwendung dieser Tools besteht jedoch der Nachteil, dass sowohl ISG-virtuos als auch industrialPhysics keine offenen Standards für einfache Integration dynamischer Modelle, wie beispielsweise das Functional Mockup Interface, unterstützen. Das bedeutet insbesondere, dass Modelle aus vorherigen Entwicklungsphasen, sofern sie nicht ursprünglich in diesen Tools erstellt wurden, nicht wiederverwendet werden können. Stattdessen muss das Modell erneut erstellt werden, was einer durchgängigen Entwicklungsmethodik, wie sie in Abschnitt 3.1.3

[1] `http://www.isg-stuttgart.de/de/isg-virtuos/virtuos.html`
[2] `http://www.machineering.de/produkte/simulation-software.html`

beschrieben wird, widerspricht. Gerade diese Durchgängigkeit im Entwicklungsprozess zur Vermeidung von Re-Implementierungen ist jedoch für eine effiziente Entwicklung unverzichtbar.

Neben den zuvor genannten kommerziellen Werkzeugen kann für die virtuelle Inbetriebnahme unter Echtzeitbedingungen die in Abschnitt 3.3 beschriebene Toolchain verwendet werden. Mit Hilfe der Toolchain ist es möglich, sowohl das Modell des Reglers als auch das Modell der Regelstrecke, welches bereits aus der Auslegungsphase vorhanden ist, auf der Steuerungshardware auszuführen. In diesem Fall wird sowohl für den Regler als auch die Regelstrecke ein eigener Funktionsbaustein angelegt. Innerhalb einer Steuerungsapplikation können die beiden Funktionsblöcke anschließend miteinander verbunden werden. Dadurch, dass die Simulation sowohl der Regelstrecke als auch des Reglers in Echtzeit abläuft, sind beide Komponenten der HiL-Simulation automatisch stets synchron. Diese Methode erlaubt es, auf einfache Weise eine virtuelle Inbetriebnahme durchzuführen, da sowohl das Modell des Reglers als auch das Modell der Regelstrecke automatisiert auf der Echtzeithardware ausgeführt werden. Nachteilig ist an dieser Vorgehensweise jedoch, dass die Echtzeitfähigkeit des Streckenmodells gegeben sein muss, was in der Praxis eine starke Einschränkung ist. Physikalische Streckenmodelle sind in der Regel komplexe Modelle, die zumeist eine Vielzahl von Events enthalten, z.B. bei der Berücksichtigung von Reibung, Anschlägen und ähnlichen Effekten. Diese Modelle sind jedoch, wie bereits in Abschnitt 3.3.2 ausführlich beschrieben wurde, in der Regel nicht echtzeitfähig. Gleichzeitig sind diese komplexen Modelle jedoch in der Regel notwendig, um in der Phase der Auslegung das dynamische Systemverhalten korrekt abzubilden. Es ist zu beachten, dass die Güte der Reglerparametrierung stark davon abhängt, wie gut das Simulationsmodell der Anlage das reale Verhalten abbildet.

Für die Durchführung der virtuellen Inbetriebnahme ist es nun auf der einen Seite denkbar, die Streckenmodelle zu vereinfachen, sodass eine Ausführung in Echtzeit ermöglicht wird. Dies widerspricht jedoch einem durchgängigen Engineering und führt im Allgemeinen zu in der Praxis nicht brauchbaren Parametrierungen des Reglers, sodass diese Variante grundsätzlich nicht zu bevorzugen ist. Stattdessen ist es wünschenswert, das nicht-echtzeitfähige Simulationsmodell der Regelstrecke ohne Anpassungen für die virtuelle Inbetriebnahme zu verwenden. Dies erschwert die HiL-Simulation jedoch enorm, da in diesem Fall dafür gesorgt werden muss, dass die Ausführung des Codes auf der Steuerungshardware und die Simulation der Regelstrecke synchron ablaufen. Auf diese Variante wird im folgenden Abschnitt eingegangen.

3.4.2 Synchronisierung von Steuerung und Simulation bei nicht-echtzeitfähiger virtueller Inbetriebnahme

Im vorherigen Abschnitt wurde mit der virtuellen Inbetriebnahme unter Echtzeitbedingungen eine Möglichkeit vorgestellt, die Steuerungsapplikation vorab, d.h. bevor

die reale Anlage verfügbar ist, zu testen. Dies setzt jedoch die Simulation der Regelstrecke in Echtzeit voraus, was in der Regel zumeist nicht ohne weiteren Aufwand, z.B. durch Vereinfachungen des Modells, möglich ist. Sollen die bereits aus der Auslegungsphase vorhandenen, komplexen, in der Regel nicht-echtzeitfähigen Modelle im Sinne einer Durchgängigkeit des Entwicklungsprozesses weiterverwendet werden, ist es notwendig, die an der Simulation beteiligten Komponenten miteinander zu synchronisieren. Nach aktuellem Stand der Technik existiert keine Simulationsumgebung, die eine derartige, nicht-echtzeitfähige HiL-Simulation ermöglicht. Im Rahmen dieser Arbeit werden zwei Ansätze vorgestellt, wie eine derartige Synchronisierung umgesetzt werden kann.

3.4.2.1 Verwendung eines Handshake-Mechanismus

Eine Möglichkeit zur Umsetzung der notwendigen Synchronisierung ist die Verwendung eines *Handshake-Verfahrens* [Menager *et al.* (2014b)]. Bei dieser Methode wird ein Taktgeber (Master) definiert, der die an der Simulation beteiligten Komponenten (Slaves) verwaltet. Die Rolle des Masters übernimmt in diesem Fall die Simulationsumgebung, während die Steuerungshardware als Slave verwendet wird. Nach dem Start der HiL-Simulation zur Durchführung der virtuellen Inbetriebnahme wird innerhalb der Simulationsumgebung zunächst ein Simulationsschritt einer vorher definierten Länge (z.B. 1 ms) ausgeführt. Anschließend wird die Simulation angehalten. Die relevanten Werte aus der Simulationsumgebung, welche als Eingangssignale für den Regler dienen, werden mit Hilfe spezieller Schnittstellenfunktionen auf die entsprechenden Variablen innerhalb der Steuerungsapplikation geschrieben. Als Schnittstelle kann beispielsweise das in Abschnitt 3.3.3.1 beschriebene Motion Logic Programming Interface verwendet werden. Die Steuerung befindet sich derweil in einem *Wartemodus*. Dieser Wartemodus wird dadurch erreicht, dass die Programme auf der Steuerung nicht zyklisch ausgeführt werden, sondern in Abhängigkeit des Wertes einer zu definierenden Variablen. Um dies zu erreichen, kann das Programm über eine *extern getriggerte Task* verwaltet werden. Dadurch wird es ermöglicht, die Ausführung eines Steuerungsprogramms aus der Simulationsumgebung heraus zu beeinflussen, indem die entsprechende Variable gesetzt wird oder nicht.

Nachdem die Eingangswerte aus der Simulationsumgebung auf die Steuerung übertragen wurden, wird aus der Simulationsumgebung heraus die Triggervariable gesetzt. Die Task detektiert den Wert der Variablen und startet die einmalige Ausführung der durch die Task verwalteten Programme. Anschließend wird der Wert der Triggervariablen wieder zurückgesetzt. Während der Berechnung des Algorithmus auf der Steuerung wartet die Simulationsumgebung auf die Beendigung des Rechenschrittes. Die vollständige Durchführung des Rechenschrittes wird mit Hilfe einer zweiten Variablen überwacht. Ist die Berechnung abgeschlossen, werden die Ausgangswerte des Reglers aus der Simulationsumgebung heraus, erneut unter Verwendung entsprechender Schnittstellenfunktionen der Steuerungshardware, in die

Simulation übertragen. Anschließend wird die Simulation für einen Schritt mit der zuvor eingestellten Schrittweite fortgesetzt.

Für eine Synchronisierung ist es zwingend erforderlich, dass die Schrittweite innerhalb der Simulationsumgebung mit der eingestellten Zykluszeit der Task auf der Steuerung übereinstimmt, sodass die jeweils durchgeführten Rechenschritte in der Simulation und auf der Steuerung gleich groß sind. Diese Methode hat den Vorteil, dass sie sehr einfach zu implementieren ist. Gleichzeitig weist diese Methode einige wesentliche Nachteile auf. Wird eine Motion-Logic-Steuerungshardware wie beispielsweise die IndraControl XM22 oder die IndraControl VPB40.3 verwendet, die neben klassischen SPS-Funktionalitäten zusätzlich Motion-Befehle zur Bewegungssteuerung von Antrieben bereitstellt, kann die Synchronisierung mit Hilfe des Handshake-Mechanismus nicht angewendet werden. Intern ist eine derartige Hardware aus zwei unabhängigen Komponenten aufgebaut, einem Logik-Teil, in dem die SPS-Applikation erstellt wird, sowie einem Motion-Teil, in dem die Antriebe angelegt werden und die Bewegungssteuerung realisiert wird. Das zuvor beschriebene Handshake-Verfahren basiert darauf, dass das Programm solange nicht abgearbeitet wird, wie die Triggervariable, die die Task auslöst, nicht gesetzt ist. Das bedeutet jedoch nicht, dass die Steuerung in der Zeit nicht weiterläuft. Durch eine Task lassen sich lediglich Programme des Logik-Teils verwalten. Funktionalitäten, die in dem Motion-Teil verwurzelt sind, wie beispielsweise auch die steuerungsinterne Sollwertgenerierung, können nicht mit Hilfe einer Task verwaltet werden. Die Sollwertgenerierung wird, genau wie alle anderen Motion-Funktionalitäten, von einem eigenen Event getriggert, welches von einem internen Taktgeber unter Berücksichtigung der eingestellten Motion-Zykluszeit ausgelöst wird. Auf derartige Events kann von außen mit Hilfe des Handshake-Mechanismus kein Einfluss genommen werden.

Aus den soeben genannten Gründen kann das Handshake-Verfahren daher lediglich für reine SPS-Anwendungen im Logik-Teil verwendet werden. In der hydraulischen Antriebstechnik werden in der Regel jedoch Motion-Befehle zur Kommandierung von Achsen benötigt und verwendet. Dies führt dazu, dass der Handshake-Mechanismus für die meisten Applikationen, die im Rahmen dieser Arbeit betrachtet werden, nicht verwendet werden kann. Des Weiteren sind häufig bereits vorgefertigte Regelalgorithmen für Standard-Regelungsaufgaben in der Firmware hinterlegt, die ebenfalls Motion-Funktionalitäten beinhalten. Da die Parametrierung dieser bereits in der Firmware enthaltenen Regler abhängig von der konkreten Regelungsaufgabe aufwendig sein kann, ist es wünschenswert, die Parametrierung derartiger Regler ebenfalls vorab an einer virtuellen Regelstrecke durchführen und testen zu können. Eine alternative Methode der Synchronisierung, die neben dem Logik-Teil zusätzlich für den Motion-Teil verwendet werden kann, wird im nächsten Abschnitt vorgestellt.

3.4.2.2 Integration eines getriggerten Modus in die Steuerungsfirmware

Der Nachteil des zuvor vorgestellten Handshake-Verfahrens ist, dass lediglich der Logik-Teil der Steuerung in der Ausführung beeinflusst werden kann. Dazu wird die

Steuerungshardware nicht angehalten, die entsprechende Task wird lediglich nicht zyklisch aufgerufen, sondern in Abhängigkeit von einer Variablen gestartet. In Echtzeit im Motion-Teil der Steuerung ablaufende Prozesse, wie beispielsweise die Generierung von Sollwerten für angeschlossene Achsen, laufen weiterhin in Echtzeit. Die Steuerung des Motion-Teils erfolgt mit Hilfe von steuerungsinternen Events, die unmittelbar vom Echtzeittaktgeber generiert werden. Im Fall der im Rahmen dieser Arbeit eingesetzten Steuerungen existieren zwei unterschiedliche steuerungsinterne Events, ein Motion-Event, an welches beispielsweise die Sollwertgenerierung gekoppelt ist, sowie ein spezifisches Bus-Event, welches die Austauschzeitpunkte der notwendigen Daten zwischen der Steuerungshardware und den Busteilnehmern, z.B. Sensoren oder Antriebe, vorgibt. Bei Verwendung des Handshake-Verfahrens geht die Synchronisierung der Simulationsumgebung mit dem Motion-Teil der Steuerung verloren, da die beiden Events unmittelbar vom Echtzeittaktgeber zyklisch ausgelöst werden.

Eine Synchronisierung der HiL-Simulation ist in diesem Fall lediglich durch ein vollständiges Entkoppeln der gesamten Steuerung von der Ausführung in Echtzeit zu realisieren. Eine derartige Funktionalität ist bisher in keiner Steuerungsfirmware verfügbar, sodass eine synchronisierte HiL-Simulation mit einem nichtechtzeitfähigen Simulationsmodell nach derzeitigem Stand der Technik nicht umsetzbar ist. Im Rahmen dieser Arbeit wurde daher eine Methode entwickelt, die dies ermöglicht. Damit die Entkopplung des Steuerungskerns von der Echtzeitausführung sowie die gezielte Triggerung von Berechnungen ebenfalls von außen gesteuert werden kann, wird diese Funktionalität in das Motion Logic Programming Interface (vgl. Abschnitt 3.3.3.1) integriert.

Insgesamt besteht die Schnittstelle aus zwei Methoden, die in der MLPI-Bibliothek verfügbar gemacht sind. Mit Hilfe der ersten Methode kann die *externe Triggerung der Steuerungsfirmware* konfiguriert werden. Innerhalb der Konfiguration wird entschieden, welche internen Events von der zyklischen Ausführung im Echtzeittakt entkoppelt werden sollen (Motion- und/oder Bus-Event). Weiterhin kann konfiguriert werden, wie nach einem Aufruf der Triggerung die Tasks abgearbeitet werden. Zum einen ist es möglich, diese unverzüglich abzuarbeiten oder auf den nächsten Takt des Echtzeittaktgebers zu warten, um sich mit diesem zu synchronisieren. Nach der Durchführung der Konfiguration befindet sich die Steuerung in einem *getriggerten Modus*, sodass die internen Events nicht länger vom Echtzeittaktgeber getriggert werden, sofern diese innerhalb der Konfiguration von der zyklischen Ausführung entkoppelt wurden. Ein Aufruf der zweiten implementierten Methode ermöglicht es schließlich, die internen Events auszulösen. Der Aufruf der Funktion erfordert als Argument das Event, welches getriggert werden soll (Motion-Event bzw. Bus-Event) sowie die Anzahl der Ausführungen pro Triggerung. Dadurch ist es möglich, das entsprechende Event beim Aufruf der Methode einfach oder mehrfach auszulösen. Der Ablauf der Synchronisation ist grundsätzlich identisch zu dem Vorgehen bei Verwendung des Handshake-Verfahrens. Nachdem die Simulation inner-

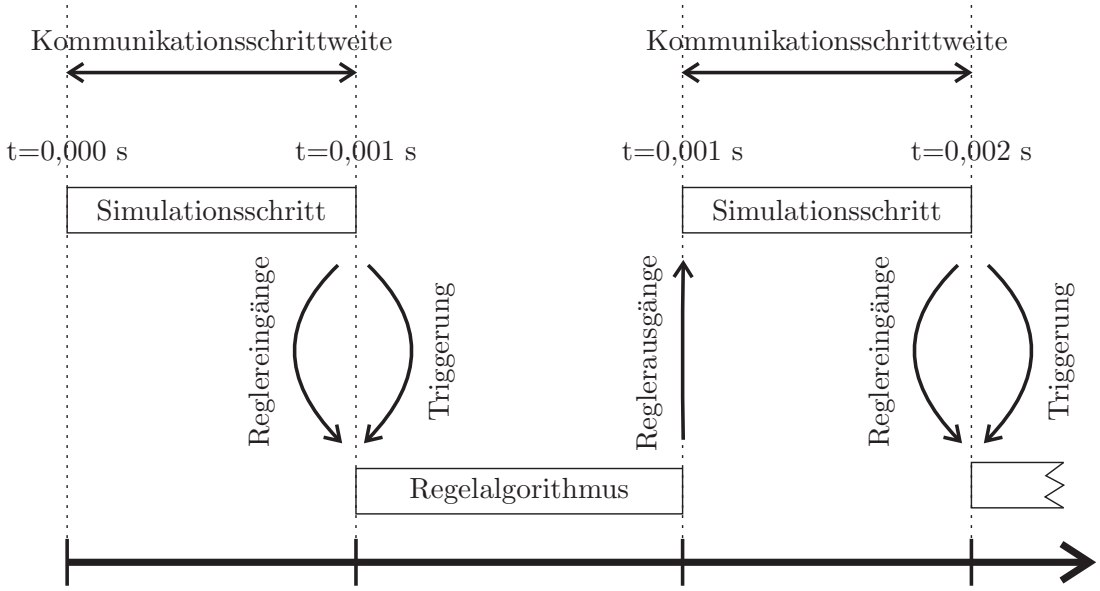

Abbildung 3.14: Synchronisierung zwischen Simulationsumgebung und Steuerung bei einer HiL-Simulation mit einer verwendeten Kommunikationsschrittweite von 1 ms

halb der Simulationsumgebung um eine gewisse Zeit fortgeschritten ist, wird diese angehalten und die Werte der Reglereingänge aus der Simulation auf die Steuerung geschrieben. Anschließend werden sämtliche internen Events einmalig ausgelöst, sodass die Berechnung eines Zyklus erfolgt. Die Ausgangswerte des Reglers werden zurück in die Simulationsumgebung transferiert und die Simulation fortgesetzt. Das Vorgehen ist in Abbildung 3.14 dargestellt. Die in diesem Abschnitt vorgestellte Methode, eine synchronisierte Hardware-In-The-Loop-Simulation zwischen einer echtzeitfähigen Steuerungshardware und einer nicht-echtzeitfähigen Simulationsumgebung durch Entkopplung der Steuerung vom internen Echtzeittaktgeber zu realisieren, wurde im Rahmen dieser Arbeit zum Patent angemeldet[1].

Zusammenfassung und Fazit Mit der zuvor beschriebenen Funktionalität der externen Triggerung der Steuerungsfirmware ist es möglich, ein beliebig komplexes, nicht-echtzeitfähiges Simulationsmodell mit einer Industriesteuerung derart zu koppeln, dass beide Komponenten stets synchronisiert sind. Dies ermöglicht einen maximalen Grad an Durchgängigkeit, da das in der Auslegungsphase erstellte Modell der Regelstrecke ohne Modifikationen für die virtuelle Inbetriebnahme verwendet werden kann. Darüber hinaus bietet die Verwendung komplexer Modelle den Vorteil, dass diese das reale Verhalten im Vergleich zu vereinfachten, echtzeitfähigen Modellen, wie sie bisher benötigt werden, deutlich besser abbilden, wodurch eine signifi-

[1]Das Patent „Verfahren zum Simulieren einer Maschine“ wurde am 13.01.2015 unter dem Aktenzeichen 10 2015 200 300.0 beim Deutschen Patent- und Markenamt angemeldet und am 14.07.2016 offengelegt.

kante Steigerung der Qualität der Parametrierung ermöglicht wird. Die Ergebnisse, die durch eine derartige virtuelle Inbetriebnahme erzielt werden, sind dennoch identisch mit den Ergebnissen, wie sie in einer Echtzeitsimulation erzielt werden würden, da der Ablauf der HiL-Simulation auch in diesem Fall konsistent und deterministisch ist. Dies liegt daran, dass die Simulation und die Steuerungsapplikation zwar nicht in Echtzeit, aber zu jedem Zeitpunkt synchronisiert ausgeführt werden. Da auch bei der Aktivierung des getriggerten Modus sämtliche Überwachungsfeatures der Steuerung, insbesondere die Funktionen zur Messung der Ausführungsdauer der Steuerungsapplikation, aktiviert bleiben, lassen sich auch in diesem Fall Zykluszeitüberschreitungen erkennen, da weiterhin jeder durchgeführte Zyklus überwacht wird. Die vorgestellte Methode ist daher für eine virtuelle Inbetriebnahme sehr gut geeignet.

3.4.2.3 Umsetzung der HiL-Simulation in der Simulationsumgebung

Für die Umsetzung der Hardware-In-The-Loop-Simulation zur Durchführung der virtuellen Inbetriebnahme ist es notwendig, die zuvor vorgestellten Methoden aus der Simulationsumgebung heraus zu verwenden. Wie bereits zuvor beschrieben, übernimmt die Simulationsumgebung die Aufgabe des Masters, während die Steuerungshardware als Slave fungiert und von der Simulationsumgebung getriggert wird. Für den Datenaustausch zwischen den beteiligten Komponenten der Hardware-In-The-Loop-Simulation wurde im Rahmen dieser Arbeit eine Modelica-Komponente *HiL-Coupler* entwickelt. Der HiL-Coupler wird in das Simulationsmodell integriert und ermöglicht die Durchführung der HiL-Simulation auf einfache Weise. Dafür baut die Komponente zu Simulationsbeginn die Verbindung zu der angeschlossenen Steuerungshardware auf und entkoppelt die Steuerung von der Ausführung in Echtzeit. Während der Simulation überträgt sie automatisch die Werte zur Steuerung, triggert die steuerungsinternen Events und überträgt die Werte zurück in die Simulation. Für die Übertragung der Signale stehen *I/O-Block*-Komponenten bereit, die an den HiL-Coupler angeschlossen werden. Die Zuordnung der Signale in der Simulation zu den Variablen in der Steuerungsapplikation erfolgt über die Variablennamen, die in den I/O-Block eingetragen werden. Für den Aufbau der Verbindung zur Steuerungshardware ist im HiL-Coupler die IP-Adresse des Targets anzugeben. Sowohl die HiL-Coupler Komponente als auch die I/O-Block Komponente ist Teil der frei erhältlichen Modelica-Bibliothek mlpi4Modelica.

Die Verwendung des HiL-Couplers in einem Modell ist in Abbildung 3.15 sichtbar. Die Komponente ersetzt den Regler im Modell und überträgt mit Hilfe des angeschlossenen I/O-Blocks insgesamt vier Signale. Davon stellen drei Signale, die Position des Zylinders sowie die beiden Drücke in den Kammern A und B des Zylinders, Eingangsgrößen des Reglers dar. Diese werden an den Regler übertragen. Nach der Durchführung der Berechnung auf der Steuerung wird das Stellsignal für das Ventil in die Simulation übertragen und mit dem entsprechenden Anschluss des Ventils verbunden. Der Aufbau der Verbindung zur Steuerung sowie die Gewährleistung

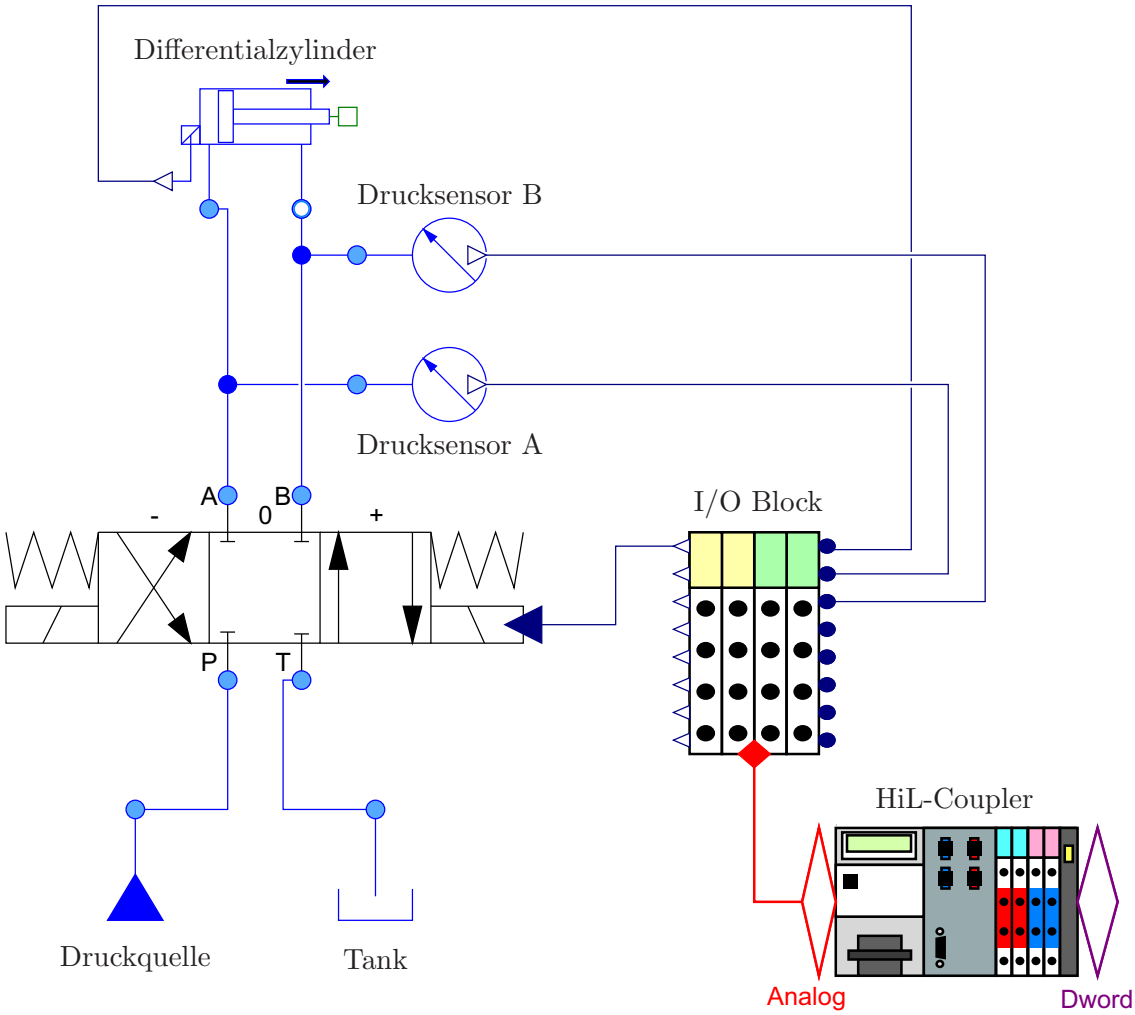

Abbildung 3.15: Verwendung der Komponente HiL-Coupler innerhalb eines Modells zur Realisierung einer HiL-Simulation

der Synchronisierung während der HiL-Simulation wird automatisch von dem HiL-Coupler übernommen. Insgesamt ist die Einbindung einer realen Steuerung in die Simulation daher auf einfache Weise möglich. Eine Anpassung der Steuerungsapplikation für die Verwendung innerhalb der HiL-Simulation ist nicht notwendig.

KAPITEL 4

Anwendung auf Beispiele

Der im vorherigen Kapitel beschriebene modellbasierte Entwicklungsprozess sowie die Methoden zur Umsetzung werden in diesem Kapitel in der Praxis angewendet. Dafür wird in einem ersten Beispiel die modellbasierte Entwicklung eines Gleichlaufs zweier hydraulischer Achsen, einem Standardfall in der hydraulischen Antriebstechnik, durchgeführt. In einem zweiten Beispiel wird eine komplexe Flaschenabfüllanlage modellbasiert entwickelt und virtuell in Betrieb genommen. An diesem Beispiel wird verdeutlicht, wie die im Rahmen dieser Arbeit entwickelten Methoden auf Basis offener Standards auch in kommerzielle Entwicklungsumgebungen integriert werden können.

4.1 Modellbasierte Entwicklung einer hydraulischen Presse

In der Praxis basieren zahlreiche Industrieanlagen auf hydraulischen Antrieben. Diese kommen häufig dann zum Einsatz, wenn eine hohe Leistungsdichte sowie das Aufbringen hoher Kräfte und Momente gefordert sind. Als Beispiel für eine hydraulische Industrieanlage wird in diesem Abschnitt eine Presse betrachtet. Diese besteht hauptsächlich aus zwei hydraulischen Achsen, die den Pressstempel bewegen. Für einen sauberen Pressvorgang ist es zwingend notwendig, dass sich beide Achsen vollständig synchron bewegen. Für die Regelung eines solchen Systems wird eine Gleichlaufregelung eingesetzt. Anhand dieses Beispiels wird gezeigt, wie eine durchgängige, modellbasierte Entwicklung auf Basis von offenen Standards in der Praxis durchgeführt werden kann. Die Auslegung und Inbetriebnahme basiert auf

den in Kapitel 3 entwickelten Methoden. Aus diesem Grund wird zunächst ein Simulationsmodell der Anlage erstellt, welches aus einem Modell der Regelstrecke sowie einem Modell eines geeigneten Reglers besteht. Unter Vorgabe eines Sollverhaltens der Anlage wird das System mit Hilfe des in Abschnitt 3.2 entwickelten Optimierungsfeatures automatisiert ausgelegt. Dabei wird eine Kombination aus Ventilen und Reglerparametern derart ermittelt, dass das vorgegebene Verhalten, welches durch ein Positionsprofil des Pressstempels beschrieben wird, bestmöglich erreicht wird. Anschließend wird die in Abschnitt 3.3 beschriebene Toolchain verwendet, um ausführbaren Code aus dem Simulationsmodell des Reglers zu generieren, der anschließend in die Steuerungsapplikation auf der realen Steuerungshardware integriert wird. Die Funktionsweise des Regelalgorithmus auf der Steuerungshardware wird zunächst mit Hilfe des Simulationsmodells der Regelstrecke in Form einer Hardware-In-The-Loop-Simulation erprobt und optimiert, bevor die Steuerungshardware an der realen Anlage verwendet wird. Die einzelnen Schritte des Entwicklungsprozesses werden in den nachfolgenden Abschnitten ausführlich beschrieben.

Bei näherer Betrachtung der zuvor genannten Vorgehensweise werden die wesentlichen Vorteile der modellbasierten Entwicklung deutlich. Die Generierung von ausführbarem Code aus der Modellbeschreibung des Reglers erspart bisher notwendige Re-Implementierungen und sorgt für eine durchgängige Wiederverwendung bereits vorhandenen Wissens. Durch die Einbindung der Hardware in die Simulationsumgebung und die sich daraus ergebende Möglichkeit einer virtuellen Inbetriebnahme kann weiterhin bereits mit dem Testen und Optimieren der Reglersoftware begonnen werden, bevor die Anlage real existiert. In einer sicheren Umgebung kann die Funktionsweise des Regelalgorithmus validiert werden. Besonders zu beachten ist, dass die HiL-Simulation auf einem Standardrechner durchgeführt wird und eine bisher in der Regel notwendige Echtzeithardware nicht benötigt wird. Die in Abschnitt 3.4.2.2 entwickelte Methode zur Wahrung der Synchronität zwischen der Steuerungshardware sowie der Simulationsumgebung bei der Einbindung der Steuerungshardware in den Simulationsablauf ermöglicht es, trotz der Nicht-Echtzeitfähigkeit der Simulation dieselben korrekten Ergebnisse zu erhalten, wie sie bisher lediglich durch eine Echtzeitsimulation mit spezieller Hardware erzielbar wären. Durch die Verwendung von Optimierungsmethoden in der Phase der Auslegung des Systems kann zudem eine von der Expertise des Ingenieurs unabhängige optimale Kombination aus Komponenten und Parametern erreicht werden. Dadurch lässt sich die Auslegung mittels trial-and-error vermeiden, wodurch die Auslegungsphase wesentlich vereinfacht wird.

Da der gesamte Entwicklungsprozess durch Modelle begleitet wird, die in der offenen, toolunabhängigen Modellierungssprache Modelica erstellt sind, und die in dieser Arbeit entwickelten Methoden auf der freien OpenModelica-Umgebung basieren, steht die vorgestellte Lösung insbesondere auch kleinen und mittleren Unternehmen zur Verfügung.

4.1.1 Modellbildung des Gesamtsystems

Nachdem für die zu entwickelnde Anlage ein Lastenheft sowie ein Pflichtenheft zusammen mit dem Kunden erstellt wurde und auf diese Weise die exakten Anforderungen festgehalten sind, besteht der erste Schritt bei der Entwicklung des Systems in der Erstellung eines Simulationsmodells der Anlage. Das Simulationsmodell besteht aus einem Modell der Regelstrecke sowie einem Modell des Reglers. Die beiden Modelle werden nachfolgend beschrieben.

4.1.1.1 Modellbildung der Regelstrecke

Das Modell der Regelstrecke, der hydraulischen Presse, besteht aus zwei separaten hydraulischen Achsen. Die Hydraulik-Komponenten sind der *HydrauLib* entnommen, einer in Modelica geschriebenen Hydraulikbibliothek der Bosch Rexroth AG. Diese Bibliothek enthält sowohl generische Komponenten, die frei parametriert werden können, als auch spezielle, bereits vorparametrierte Rexroth-Komponenten. Das dynamische Verhalten ist in diesen Komponenten bereits fest implementiert. Jede Achse der hydraulischen Presse besteht aus einem Differentialzylinder, einem Ventil zur Regulierung des Volumenstroms der Hydraulikflüssigkeit sowie Leitungen zwischen Ventil und Zylinder. Zur Messung des Drucks in den beiden Zylinderkammern stehen pro Achse zwei Drucksensoren zur Verfügung. Die Erfassung der jeweiligen Zylinderposition ist mit Hilfe des in dem Zylinder integrierten Wegmesssystem möglich. Weiterhin beinhaltet das Modell eine Konstantdruckquelle und einen Tank zur Versorgung mit Hydraulikflüssigkeit sowie eine Fluidkomponente, die das Fluidmodell beinhaltet. Das in der Sprache Modelica erstellte Modell der Regelstrecke ist in Abbildung 4.1 abgebildet. Die wesentlichen Eigenschaften des Zylindermodells sowie des Ventilmodells werden im Folgenden kurz erläutert.

Zylindermodell Bei dem Zylindermodell handelt es sich um ein generisches Modell eines Differentialzylinders, welches sowohl ein Reibungsmodell als auch ein Anschlagsmodell beinhaltet. Weiterhin ist eine interne Leckage, d.h. ein Ölfluss zwischen den beiden Zylinderkammern über den Kolben, sowie eine externe Leckage, d.h. ein Ölfluss entlang der Kolbenstange aus dem Zylinder heraus, berücksichtigt.

Ventilmodell Für die Abbildung des Ventils werden im Rahmen dieser Arbeit spezielle, vorparametrierte Rexroth-Ventilmodelle verwendet. Zwar handelt es sich bei dem in der HydrauLib vorhandenen Ventilmodell grundsätzlich ebenfalls um ein generisches Modell, im Rahmen der Bibliothek wurden diese jedoch bereits so parametriert, dass sich das Verhalten des realen Ventils ergibt. Die Parametrierung basiert hierbei auf den Daten und Kennlinien des Datenblattes. Bei der Modellierung wird der spezielle Kontext der Systemsimulation berücksichtigt, d.h. der Fokus liegt auf einer Modellierung derart, dass das grundsätzliche dynamische Verhalten des Ventils im Systemkontext abgebildet wird. Die Dynamik des Ventils ist dementsprechend vereinfachend durch ein PT3-Verhalten angenähert. Neben der Ventildynamik sind

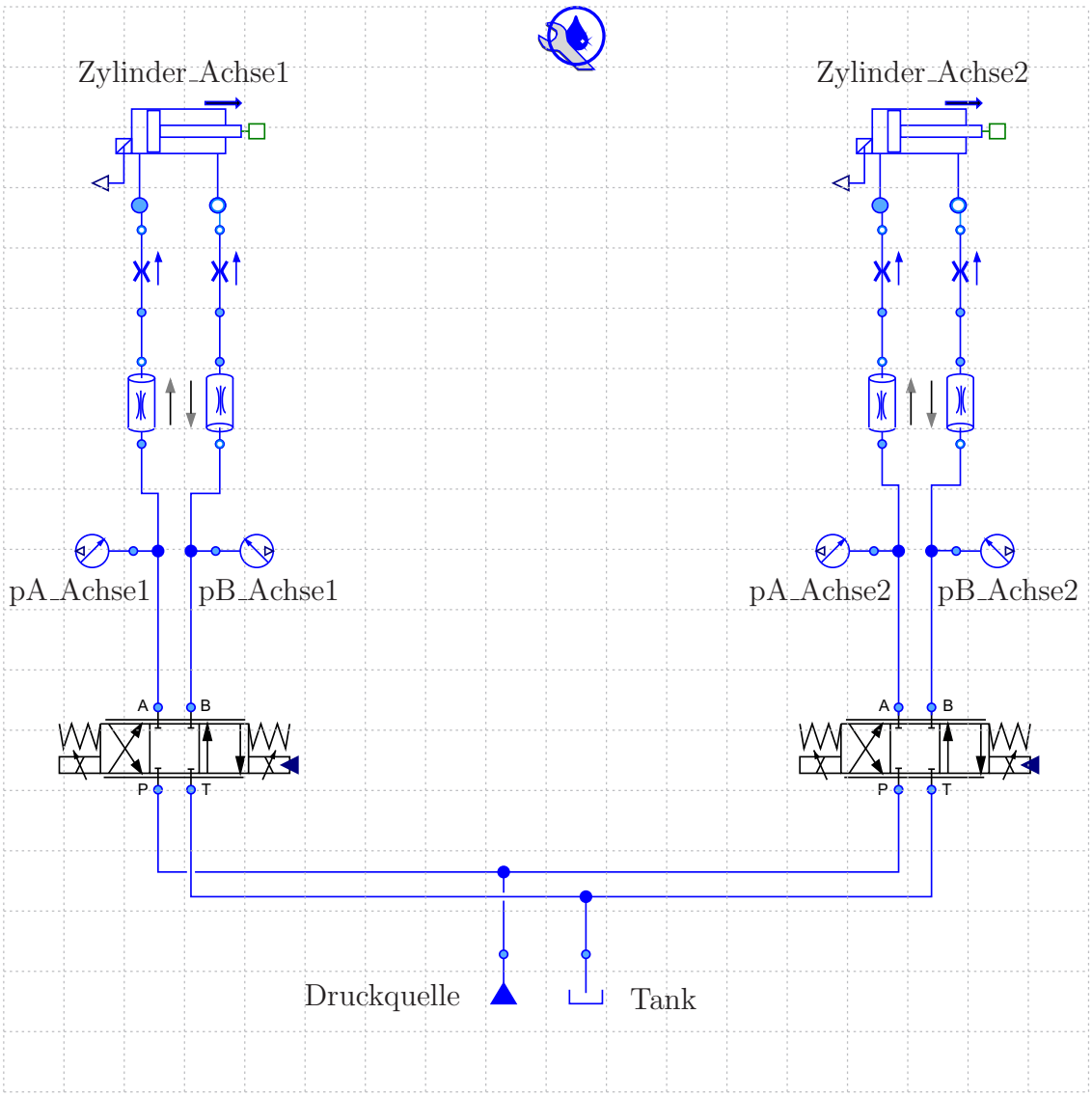

Abbildung 4.1: Modelica-Modell der Regelstrecke

in dem Modell die Leistungsgrenze des Ventils sowie Leckagen berücksichtigt. Die Durchflusscharakteristik wird entweder über die Blendengleichung beschrieben oder über Kennlinien angegeben.

4.1.1.2 Modellbildung des Reglers

Die Regelung des Systems erfolgt mit Hilfe eines Gleichlaufreglers, der dafür sorgt, dass sich die beiden Zylinder stets synchron bewegen. Er besteht aus je einem *Motion-Regler* für die beiden hydraulischen Achsen und einem *Sync-Regler*. Die Gesamtstruktur des Reglers ist in Abbildung 4.2 dargestellt.

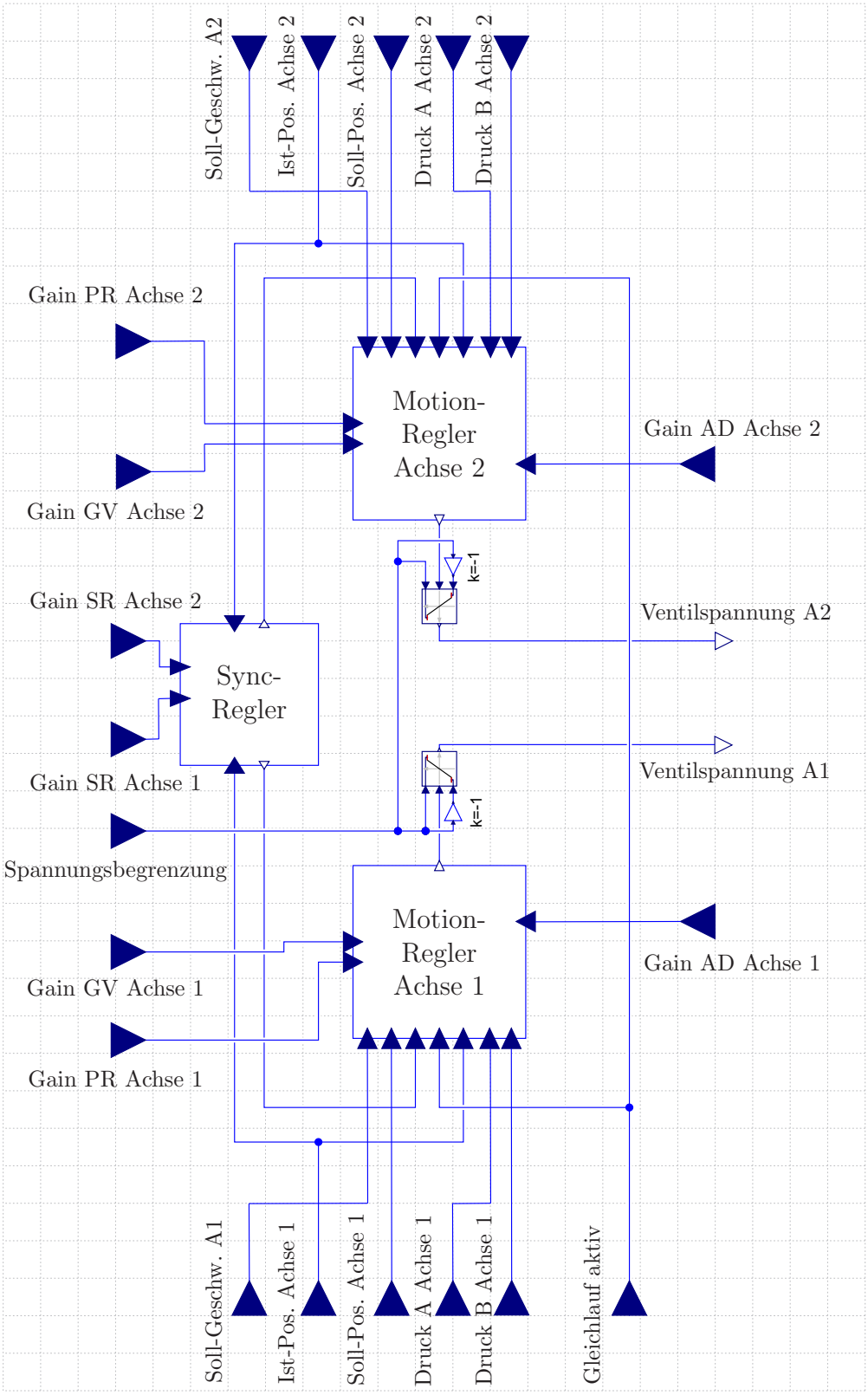

Abbildung 4.2: Modelica-Modell des Gleichlaufreglers

Der Motion-Regler beinhaltet eine Positionsregelung mit Geschwindigkeitsvorsteuerung sowie eine aktive Dämpfung und hat die Aufgabe, den Zylinderkolben einem vorgegebenen Sollwert möglichst gut nachzuführen. Für die Regelung werden die aktuelle Istposition, die Sollposition (für die Positionsregelung) sowie die Sollgeschwindigkeit (für die Geschwindigkeitsvorsteuerung) und die beiden Drücke in den Zylinderkammern (für die aktive Dämpfung) benötigt. Das Verhalten des Reglers kann weiterhin mit Hilfe von drei freien Reglerparametern (Verstärkungsfaktor für die Positionsregelung (*Gain PR*), Verstärkungsfaktor für die Geschwindigkeitsvorsteuerung (*Gain GV*) sowie Verstärkungsfaktor für die Aktive Dämpfung (*Gain AD*)) beeinflusst werden. Diese drei Regelungsparameter stehen getrennt für jede Achse zur Verfügung, sodass beide Achsen unabhängig voneinander parametriert werden können, um beispielsweise fertigungsbedingte Toleranzen in den beiden Achsen ausgleichen zu können. Mit Hilfe der beiden Motion-Regler wird folglich erreicht, dass beide Achsen unabhängig voneinander der vorgegebenen Sollposition folgen.

Der Sync-Regler sorgt zusätzlich dafür, dass sich die beiden unabhängigen Achsen synchron zueinander bewegen. Dafür werden die beiden Istpositionen der Achsen benötigt. Innerhalb des Sync-Reglers wird zunächst der Mittelwert der beiden Istpositionen gebildet. Anschließend wird für jede Achse ein Korrekturwert des Positionssollwertes basierend auf der Differenz zwischen tatsächlicher Position der Achse und dem zuvor berechneten Positionsmittelwert beider Achsen berechnet. Dieser Korrekturwert kann anschließend mit Hilfe eines Verstärkungsfaktors für jede Achse individuell gewichtet werden. Die resultierenden Werte für beide Achsen werden den beiden Motion-Reglern über den jeweiligen Eingang zugeführt. Abhängig davon, ob die tatsächliche Position höher oder niedriger ist als der Mittelwert, ergibt sich ein unterschiedliches Vorzeichen für die Anpassung. Der Regler passt auf diese Weise die Positionssollwerte beider Achsen abhängig von der aktuellen Positionsabweichung an, was zu einer Synchronisierung der Achsen führt. Zur Parametrierung des Sync-Reglers stehen zwei freie Reglerparameter (Verstärkungsfaktor des Sync-Reglers (*Gain SR*) für beide Achsen) zur Verfügung.

Auf Basis der Eingänge berechnet der Gleichlaufregler die Ventilstellgrößen beider Achsen. Über einen weiteren Eingang *Gleichlauf aktiv* lässt sich zudem der Sync-Regler abschalten, um im Rahmen der Inbetriebnahme zunächst beide Achsen unabhängig voneinander verfahren zu können. Die Ventilstellgröße kann zudem mit Hilfe des Reglereingangs *Spannungsbegrenzung* auf einen vorgegebenen Wert begrenzt werden.

4.1.2 Einsatz von Optimierungsmethoden zur Auslegung

Nachdem die Strukturen der Simulationsmodelle der Regelstrecke sowie des Reglers nun zunächst allgemein festgelegt sind, besteht der nächste Schritt darin, konkrete Komponenten aus den zur Verfügung stehenden Modellbibliotheken und Reglerparameter derart auszuwählen, dass die für die Presse definierten Anforderungen

realisiert werden. Für den Betrieb der Presse steht eine Sollwertvorgabe in Form eines Profils bereit, welchem die beiden Zylinder bestmöglich folgen sollen. Der Pressstempel, welcher ein Gewicht von 400 kg besitzt, soll auf eine Position von 500 mm bezogen auf das Wegmesssystem des Zylinders bewegt werden und dort für eine Dauer von fünf Sekunden verweilen. Während dieser Zeit wird der Pressvorgang durchgeführt. Anschließend soll sich der Zylinder bis auf eine Position von 200 mm bezogen auf das Wegmesssystem des Zylinders bewegen. In dieser Warteposition verweilt die Anlage ebenfalls fünf Sekunden. In dieser Zeit wird das fertig gepresste Produkt aus der Presse entnommen und ein neuer Rohling eingelegt. Für die Bewegungen stehen Geschwindigkeits- sowie Beschleunigungsgrenzwerte bereit.

Für die Errichtung der Anlage wird ein fest definierter Differentialzylinder mit einer maximalen Länge von 900 mm, einem Kolbendurchmesser von 50 mm sowie einem Stangendurchmesser von 36 mm verwendet. Für das verwendete Ventil existiert jedoch keine Einschränkung, dieses kann frei gewählt werden. Es wird für dieses Beispiel angenommen, dass fünf verschiedene Regelventile mit unterschiedlichen Größen zur Verfügung stehen. Die Reglerparameter sind ebenfalls vollständig frei einstellbar. Zur Auslegung des Systems kann versucht werden, durch händisches Probieren eine geeignete Kombination aus Ventilen und Reglerparametern zu ermitteln. Gleichzeitig ist es jedoch auch möglich, die Auslegung unter Zuhilfenahme mathematischer Optimierungsmethoden zu automatisieren. Dazu wird das in Abschnitt 3.2 beschriebene Optimierungsmodul verwendet. Die Optimierung des Ventils erfolgt mit Hilfe der vorgestellten Optimierungskomponente der OptimLib (vgl. Abschnitt 3.2.2.2).

Das zu optimierende Gesamtsystem, welches sich aus den Modellen der Regelstrecke sowie des Reglers ergibt, ist in Abbildung 4.3 dargestellt. Für die Optimierung wird der Genetische Algorithmus verwendet, der sich für die Lösung gemischt-ganzzahliger Optimierungsprobleme, bei denen ein Komponententausch durchgeführt wird, am besten eignet. Dies liegt neben der Möglichkeit der Parallelisierung, durch die eine deutliche Verkürzung der Auslegungszeit erreicht werden kann, an den speziell für einen Komponententausch angepassten Operatoren (vgl. Abschnitt 3.2.2.1). Insgesamt ergeben sich für die Optimierung zehn freie Parameter. Diese sind zum einen die beiden Ventile sowie die acht freien Reglerparameter des Gleichlaufreglers. Diese sind in der Abbildung in grün markiert. Da es sich jedoch um zwei identische Achsen handelt und sich somit das Verhalten beider Achsen in der Simulation nicht unterscheidet, kann die simulative Auslegung auf eine Achse reduziert werden und diese für beide Achsen gleichermaßen verwendet werden. Dadurch reduziert sich die Anzahl der freien Optimierungsvariablen auf fünf. Es ist jedoch zu beachten, dass sich beide Achsen in der Praxis bei Verwendung des realen Systems bedingt durch Fertigungstoleranzen durchaus unterschiedlich verhalten können. Aus diesem Grund ist es notwendig, innerhalb des Reglers die getrennte Parametrierung beider Achsen zu gewährleisten, sodass sich diese fertigungsbedingten Unterschiede später an der realen Anlage nachträglich korrigieren lassen.

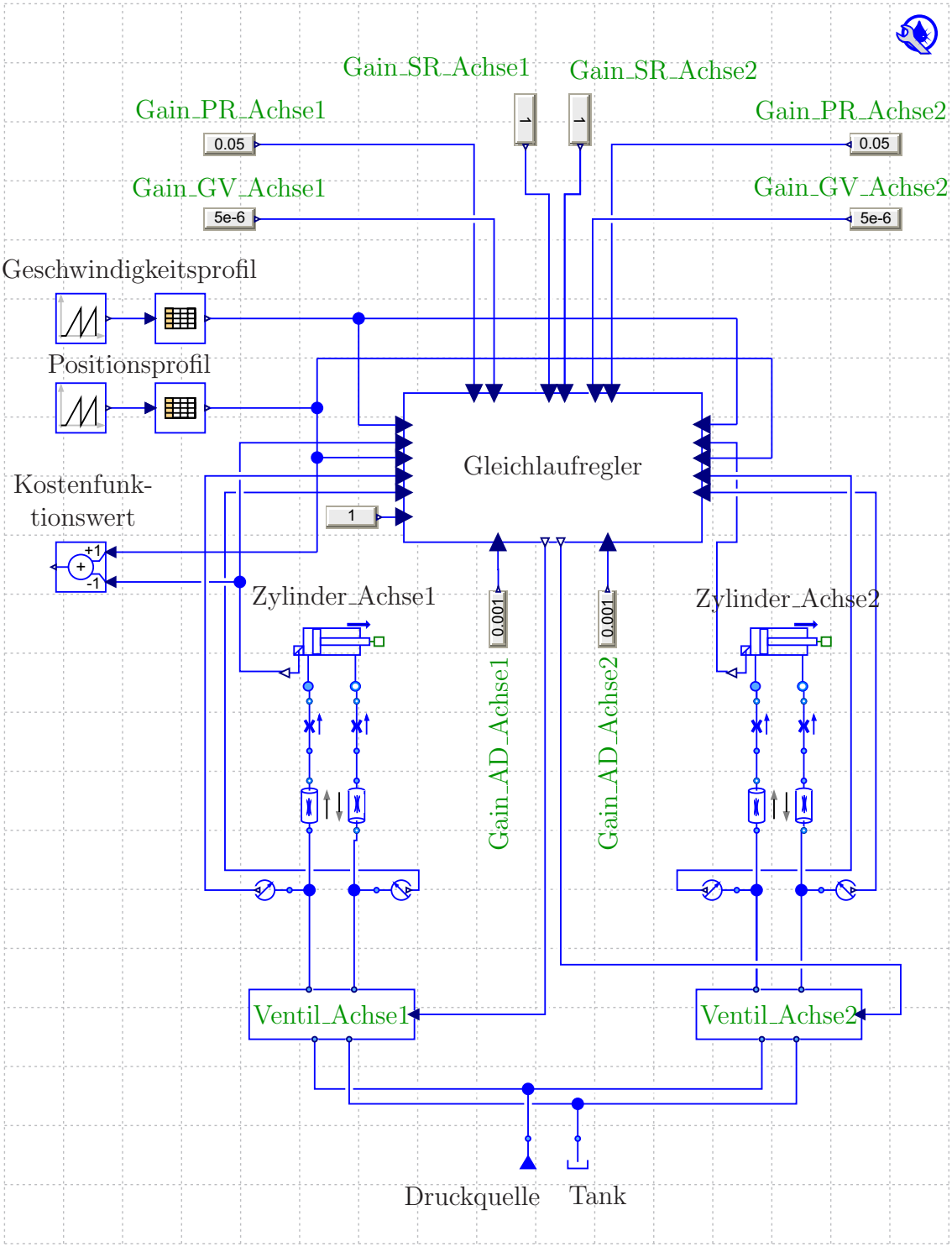

Abbildung 4.3: Gesamtmodell bestehend aus Regelstrecke und Regler für die Optimierung (freie Optimierungsvariablen sind grün gekennzeichnet)

Um die Effizienzsteigerung durch die Verwendung einer parallelen Auswertung innerhalb des Genetischen Algorithmus bewerten zu können, wird vor der Optimierung ein Benchmark der Parallelisierung durchgeführt, der im Folgenden beschrieben wird.

Benchmark der Parallelisierung Die Güte der Parallelisierung eines Programms wird in der Regel mit Hilfe des Speedups angegeben. Dieser beschreibt den Zusammenhang zwischen der benötigten Zeit bei einer seriellen Ausführung eines Programms und der benötigten Zeit bei einer parallelen Ausführung eines Programms, d.h.

$$S_p = \frac{T_1}{T_p}. \tag{4.1}$$

Hierbei beschreibt T_1 die Ausführungszeit bei Verwendung lediglich eines Prozessors, während T_p die Ausführungszeit bei Verwendung von p Prozessoren beschreibt. Im Idealfall beträgt der Speedup $S_p = p$. Dieser Wert wird in der Praxis aufgrund von vorhandenem Overhead zum Verwalten der unterschiedlichen Threads jedoch nicht erreicht. Der Speedup liegt daher im Intervall $1 \leq S_p \leq p$.

Der im Rahmen dieser Arbeit verwendete Computer ist mit einem *Intel Core i7-4810MQ* Prozessor mit vier Kernen, die jeweils eine Grundtaktfrequenz von 2,8 GHz[1] besitzen, und 32 GB Arbeitsspeicher ausgestattet. Durch die Hyperthreading Technologie sind acht Threads parallel ausführbar. Aus diesem Grund werden im Rahmen dieses Benchmarks acht unabhängige Simulationen durchgeführt, die in einem Fall sequentiell ablaufen und in dem anderen Fall auf die acht zur Verfügung stehenden Threads aufgeteilt und parallel bearbeitet werden. Da das Simulationsproblem mit verschiedenen Parametersätzen unterschiedlich schwer zu lösen sein kann, wird für diese Untersuchung eine feste Parametrierung verwendet, sodass in beiden Fällen acht identische Simulationen durchgeführt werden. Es ergeben sich für das in Abbildung 4.3 dargestellte Modell bei einer Simulationszeit von 15 s die nachfolgend dargestellten Ausführungszeiten.

Sequentielle Durchführung	694,592 s
Parallele Durchführung	234,959 s

Der sich daraus ergebende Speedup beträgt

$$S_p = \frac{694{,}592\,\mathrm{s}}{234{,}959\,\mathrm{s}} = 2{,}956. \tag{4.2}$$

[1] Mit Hilfe der Intel Turbo-Boost-Technologie kann die Taktfrequenz pro Kern auf eine maximale Taktfrequenz von 3,8 GHz gesteigert werden

Da dieser Speedup sich daraus ergibt, dass die ohnehin unabhängigen Simulationen bei der Berechnung der Fitnesswerte parallel ausgeführt werden und somit praktisch kein Aufwand zur Umsetzung einer parallelen Auswertung notwendig ist, wird im Folgenden stets eine parallele Berechnung verwendet. Es ist bei der Interpretation des Speedups zu beachten, dass der verwendete Prozessor trotz der Hyperthreading Technologie, durch die acht Threads parallel gestartet werden können, lediglich vier unabhängige Prozessoreinheiten besitzt.

Durchführung der Optimierung Zur Bestimmung der optimalen Komponenten und Reglerparameter für die in diesem Beispiel betrachteten Presse wird auf Basis des Simulationsmodells aus Abbildung 4.3 die Optimierung mit Hilfe des Genetischen Algorithmus durchgeführt. Wie bereits zuvor beschrieben, wird für beide Achsen eine identische Konfiguration verwendet, sodass das Optimierungsproblem fünf freie Optimierungsvariablen (Ventilkomponente, Verstärkungsfaktor Positionsregelung, Verstärkungsfaktor Geschwindigkeitsvorsteuerung, Verstärkungsfaktor Aktive Dämpfung, Verstärkungsfaktor Sync-Regler) beinhaltet.

Zur Vermeidung physikalisch unsinniger Parameterkombinationen wird der Wertebereich der einzelnen Optimierungsvariablen begrenzt. Dadurch reduziert sich gleichzeitig die Anzahl der benötigten Simulationen, da Teile des Wertebereiches, in denen ohnehin keine Lösung gefunden wird, nicht durchsucht werden. Das Optimierungsziel besteht in der Minimierung der Variablen *Kostenfunktionswert* in Abbildung 4.3, die die Abweichung zwischen der Sollposition, gegeben durch das Profil, und der tatsächlichen Istposition des Zylinders der ersten Achse beschreibt. Dazu wird der zeitliche Verlauf der Ergebnisgröße diskretisiert und die Summe gebildet.

In Tabelle 4.1 sind die Ergebnisse der Optimierung zusammengefasst. Neben dem ermittelten, optimalen Wert ist jeweils die obere und untere Grenze des Wertebereiches der Optimierungsvariablen dargestellt. Das Verhalten des Systems bei

Optimierungsvariable	Opt. Wert	Min. Wert	Max. Wert
Ventilkomponente	2[1]	1	5
Gain PR	0,0742	0,01	0,20
Gain GV	7,21e-6	1e-6	1e-5
Gain AD	0,0019	0,001	0,01
Gain SR	2,527	1	10

Tabelle 4.1: Ergebnisse der Optimierung für die automatische Systemauslegung ($\lambda = 100, \mu = 200$, 5 Generationen)

[1]Dies entspricht dem Regelventil 4WRSE 10 V 50

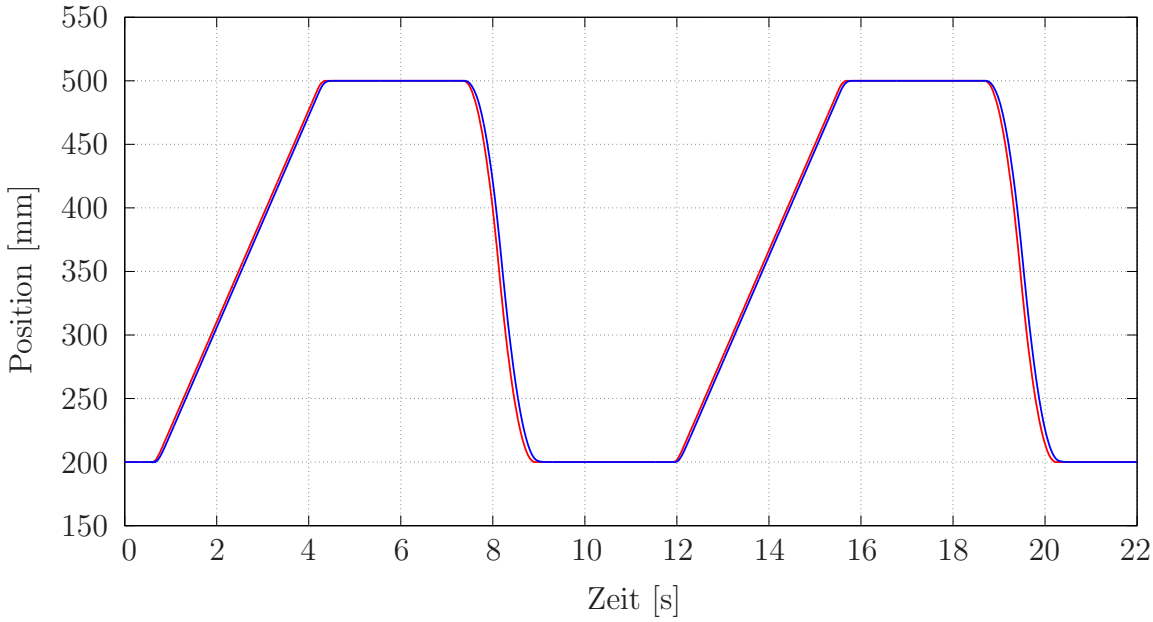

Abbildung 4.4: Verhalten des Systems nach der Optimierung (rot: Sollposition des Zylinderkolbens; blau: Istposition Zylinder Achse 1)

Verwendung der ermittelten Parameter zeigt Abbildung 4.4. Werden die beiden abgebildeten Kurven miteinander verglichen, ist eine sehr gute Übereinstimmung erkennbar.

4.1.3 Codegenerierung und virtuelle Inbetriebnahme des Reglers

Nachdem die Anlage ausgelegt ist und die verwendeten Systemkomponenten bekannt sind, wird mit der Fertigung der Anlage begonnen. Parallel dazu wird die Steuerungsapplikation auf der an der realen Anlage verwendeten Steuerungshardware implementiert. In bisherigen Entwicklungsprozessen wird in dieser Phase der Regelalgorithmus, der zuvor im Rahmen der Phase der Auslegung in der Simulationsumgebung erstellt wurde, nicht weiter verwendet und stattdessen der Regler erneut implementiert. Zusätzlich erfolgt diese Re-Implementierung zumeist in SPS-Programmiersprachen nach IEC 61131-3 und somit in einer anderen Sprache.

Im Sinne eines durchgängigen Engineerings und der damit einhergehenden Wiederverwendung bereits vorhandenen Wissens aus vorherigen Entwicklungsschritten ist es sinnvoll, den bereits vorhandenen Regelalgorithmus aus der Simulationsumgebung weiterzuverwenden. Gelingt dies, wird die Re-Implementierung vermieden, was nicht nur zu Zeiteinsparungen führt, sondern auch eine potentielle Fehlerquelle eliminiert. Um die Überführung des Reglermodells auf die Steuerungshardware umzusetzen, ist es notwendig, aus dem Modell auf der Steuerungshardware ausführbaren Code zu erzeugen und diesen auf der Steuerungshardware auszuführen.

4.1.3.1 Überführung des Reglers auf die Steuerungshardware

Die Überführung des Reglers aus der Simulationsumgebung auf die Steuerungshardware erfolgt mit Hilfe der in Abschnitt 3.3 beschriebenen Toolchain. Für eine einfache Anwendung der beschriebenen Toolchain wurde im Rahmen dieser Arbeit eine Software mit einer graphischen Bedienoberfläche entwickelt, mit der die notwendigen Schritte automatisiert durchgeführt werden können. Nach Auswahl eines Modelica-Modells wird im Hintergrund automatisch Code aus dem Simulationsmodell generiert, dieser zusammen mit dem Simulationskern (vgl. Abschnitt 3.3.2) kompiliert und die kompilierte Bibliothek auf die Steuerungshardware verschoben. Zusätzlich wird ein SPS-Funktionsbaustein generiert, mit dessen Hilfe der Code in die Steuerungsapplikation eingebettet wird (vgl. Abschnitt 3.3.3.3).

Die Generierung des Funktionsbausteins erfolgt unter Verwendung des PLCopen XML Standards [Esteez *et al.* (2009)]. Der Funktionsbaustein beinhaltet neben den Informationen, die für die Anbindung der kompilierten Bibliothek an den Funktionsbaustein benötigt werden, sämtliche Eingänge und Ausgänge des Reglers und ermöglicht es somit, auf die Variablen im Modell zuzugreifen und somit den auf der Steuerung laufenden Regler zu parametrieren. Für die Verwendung des Funktionsbausteins ist dieser in die Steuerungsapplikation, die mit der Standard-Entwicklungsumgebung IndraWorks erstellt wird, zu importieren. Das bedeutet, dass der Applikationsingenieur weiterhin in seiner vertrauten Entwicklungsumgebung arbeitet und wie gewohnt die Steuerungshardware und die verwendeten Erweiterungsmodule einrichten und konfigurieren kann. Durch Verwendung der Methode der Codegenerierung kann jedoch auf die bisher notwendige Re-Implementierung des Regelalgorithmus in SPS-Programmiersprachen nach IEC 61131-3 verzichtet werden.

Für die Verwendung des Simulationsmodells auf der realen Steuerungshardware ist es notwendig, dass dieses in Echtzeit ausgeführt wird. Wie bereits in Abschnitt 3.3.2.1 ausführlich beschrieben, ist die Wahl des numerischen Lösungsverfahrens in diesem Fall von besonderer Bedeutung. Im Rahmen dieser Arbeit ist daher der Simulationskern um echtzeitfähige Lösungsalgorithmen erweitert worden. Neben dem Expliziten Euler-Verfahren, welches für einfache, mathematisch nicht-steife Systeme verwendet werden kann, stehen linear-implizite Verfahren verschiedener Ordnung (Rosenbrock-Verfahren) zur Verfügung, die sich auch für mathematisch steife Systeme eignen. Am Beispiel der Echtzeitsimulation einer hydraulischen Achse werden diese Verfahren in [Menager *et al.* (2015a)] verglichen. Da es sich bei dem Modell der hydraulischen Achse um ein mathematisch steifes System handelt, ist die Simulation mit dem Expliziten Euler-Verfahren in diesem Fall nicht möglich. Mit den linear-impliziten Rosenbrock-Verfahren kann das Modell jedoch simuliert werden. Wie erwartet, steigt bei konstant gehaltener Schrittweite die Genauigkeit mit der Ordnung des Verfahrens. In diesem Beispiel ist es jedoch lediglich erforderlich, das Modell des Reglers auf der Steuerungshardware auszuführen. Für die Echtzeitsimulation dieses Modells ist die Verwendung des in Abschnitt 2.2.1.1 behandelten

Expliziten Euler-Verfahrens ausreichend. Die verwendete Zykluszeit der Steuerung für dieses Beispiel beträgt 1 ms. Dies entspricht gleichzeitig der zu verwendenden Schrittweite des numerischen Verfahrens.

4.1.3.2 Durchführung der virtuellen Inbetriebnahme

In der Regel besteht eine Steuerungsapplikation nicht nur aus einem einzelnen Funktionsbaustein mit einem Regler, sondern einer Vielzahl weiterer Bausteine. Dies sorgt dafür, dass diese häufig eine sehr komplexe Struktur aufweist. Neben den reinen Regelungsalgorithmen sind bei zahlreichen Anwendungen beispielsweise Sicherheitsfunktionalitäten notwendig, die im Fehlerfall die Anlage in einen sicheren Zustand versetzen. Diese werden ebenfalls der Steuerungsapplikation hinzugefügt. Um Fehler innerhalb der Steuerungsapplikation frühzeitig im Entwicklungsprozess finden zu können, ist es wünschenswert, diese nicht erst testen zu können, wenn die reale Anlage bereits aufgebaut wurde. Durch die in Abschnitt 3.4 beschriebene Methodik der virtuellen Inbetriebnahme kann der Funktionstest des Steuerungscodes unter Zuhilfenahme des aus der Phase der Auslegung bereits vorhandenen Modells der Regelstrecke bereits vorgenommen werden, bevor die reale Anlage gefertigt ist. Dazu ist es notwendig, die reale Steuerungshardware mit der Simulationsumgebung zu koppeln.

Die Integration der Steuerungshardware in die Simulationsumgebung wird mit Hilfe des HiL-Couplers (vgl. Abschnitt 3.4.2.3) realisiert. Durch die Funktionalität des Motion Logic Programming Interfaces der Steuerung, die insbesondere die Entkopplung des Motionkerns von der Ausführung in Echtzeit und die Synchronisierung mit der Simulationsumgebung ermöglicht, kann das bereits aus der Auslegungsphase vorhandene Modell der Regelstrecke unverändert wiederverwendet werden. Ausgehend von dem Gesamtmodell aus Abbildung 4.3 ist lediglich das Modell des Reglers zu entfernen und die HiL-Coupler Komponente sowie die I/O-Blöcke hinzuzufügen. Das sich ergebende Modell ist in Abbildung 4.5 gezeigt. Dieser Ansatz unterscheidet sich grundlegend von den bisher zur Verfügung stehenden Möglichkeiten einer Hardware-In-The-Loop-Simulation, die eine Anpassung der Simulationsgeschwindigkeit an den Echtzeittakt der Steuerungshardware erfordern, wodurch in der Regel das bisherige Modell aufgrund der Nicht-Echtzeitfähigkeit nicht weiterverwendet werden kann. Die notwendige Anpassung des Streckenmodells widerspricht dabei dem Konzept der durchgängigen Verwendung der Modelle im Sinne eines modellbasierten Engineerings.

Die virtuelle Inbetriebnahme ermöglicht neben der Validierung des Steuerungscodes auf der realen Steuerung sowie der damit einhergehenden frühen Detektion von Fehlern gleichzeitig das Testen von kritischen Systemzuständen. Durch Injektion von Fehlerzuständen in den Komponentenmodellen kann das Verhalten der Steuerungsapplikation auf einfache Weise untersucht werden, wodurch sich letztendlich eine bessere Produktqualität ergibt, da vorab Fehlerfälle simuliert werden können, die

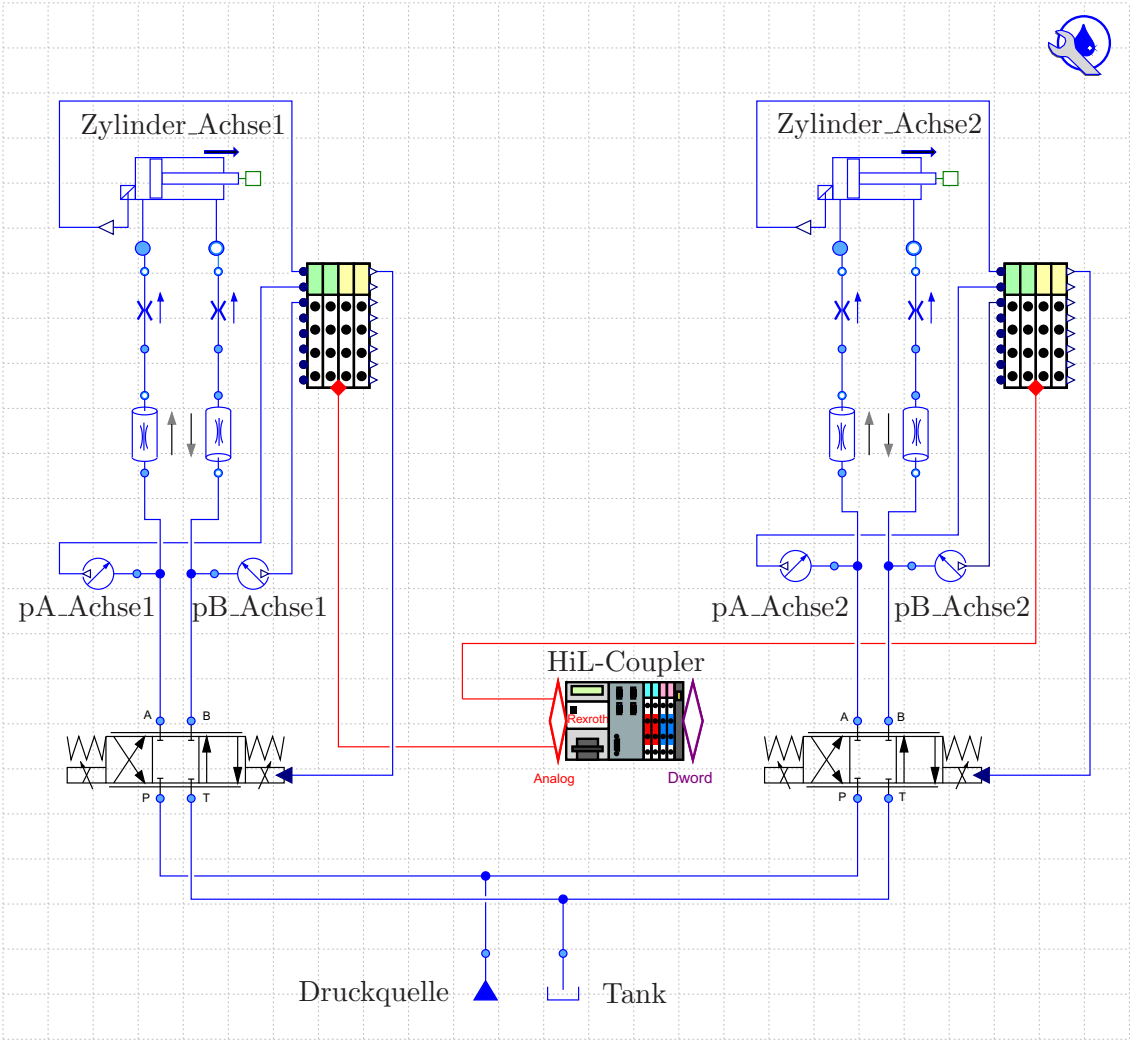

Abbildung 4.5: Modelica-Modell für virtuelle Inbetriebnahme

bei einer realen Anlage unter Umständen die mechanische Zerstörung der Anlage zur Folge hätten. Nach Abschluss der virtuellen Inbetriebnahme ist sichergestellt, dass das grundsätzliche Verhalten des Steuerungscodes mit dem gewünschten Verhalten übereinstimmt.

4.1.4 Einsatz der Steuerung an realer Anlage

Nachdem die Anlage simulativ unter Zuhilfenahme mathematischer Optimierungsverfahren ausgelegt, die Anlage gefertigt und montiert und die Steuerungsapplikation vorab auf der Steuerungshardware virtuell in Betrieb genommen wurde, folgt mit der Inbetriebsetzung der realen Anlage der letzte Schritt im Entwicklungsprozess. Durch die durchgeführte virtuelle Inbetriebnahme ist der Reglercode bereits getestet, sodass eventuell vorhandene Fehler beseitigt werden konnten. Aufgrund von vorhandenen Abweichungen zwischen dem Verhalten der realen Anlage und

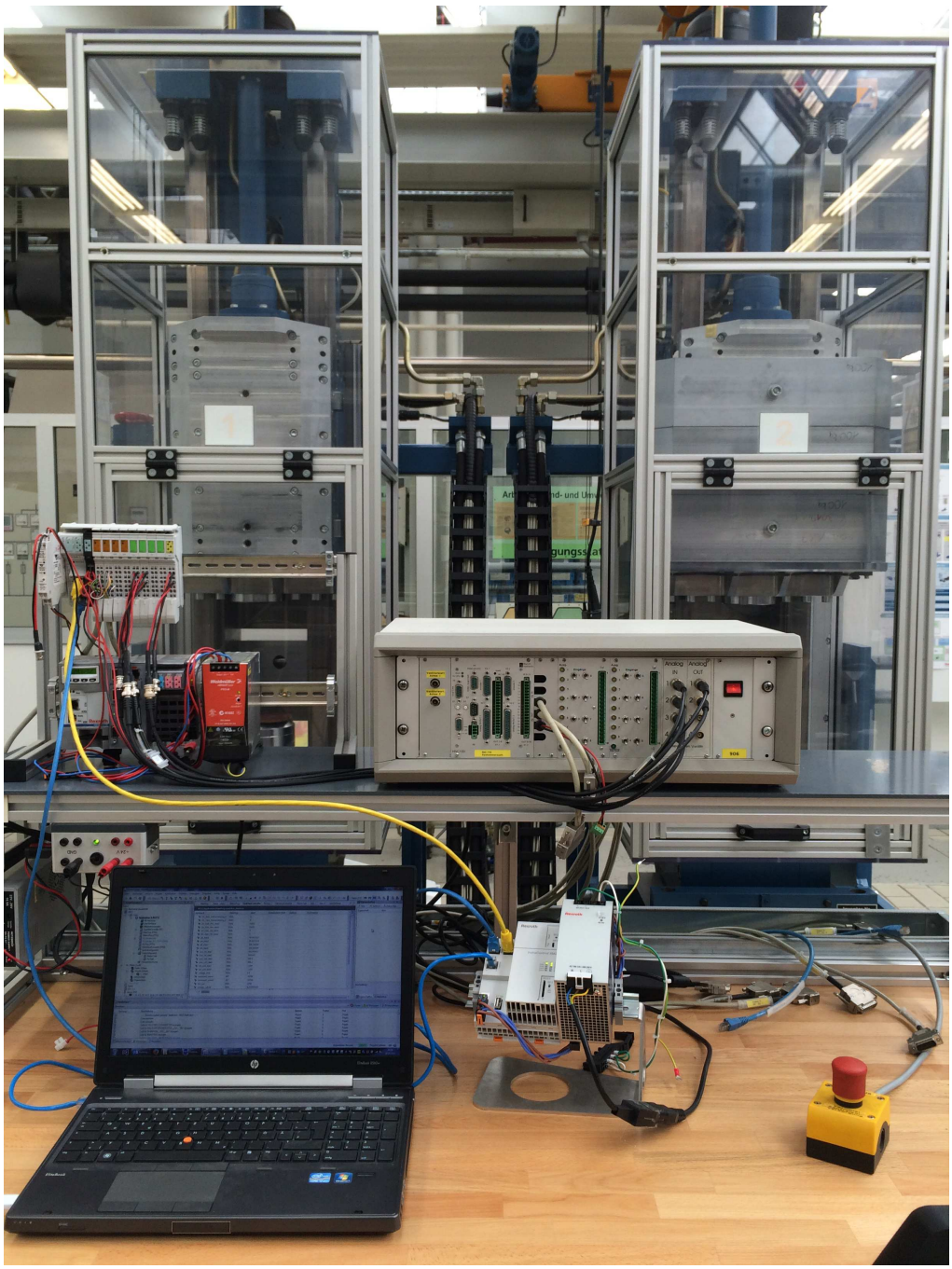

Abbildung 4.6: Aufbau des Gleichlaufprüfstandes mit zwei hydraulischen Achsen

dem Simulationsmodell der Anlage sind die zuvor ermittelten Reglerparameter für die reale Anlage zwar nicht zwangsläufig optimal, sie stellen jedoch eine zumeist sehr gute Basis dar, auf der eine letzte Feinoptimierung basieren kann. Es ist zu beachten, dass trotz der bereits durchgeführten virtuellen Inbetriebnahme die abschließende Inbetriebnahme an der realen Anlage unerlässlich ist. Im Rahmen der

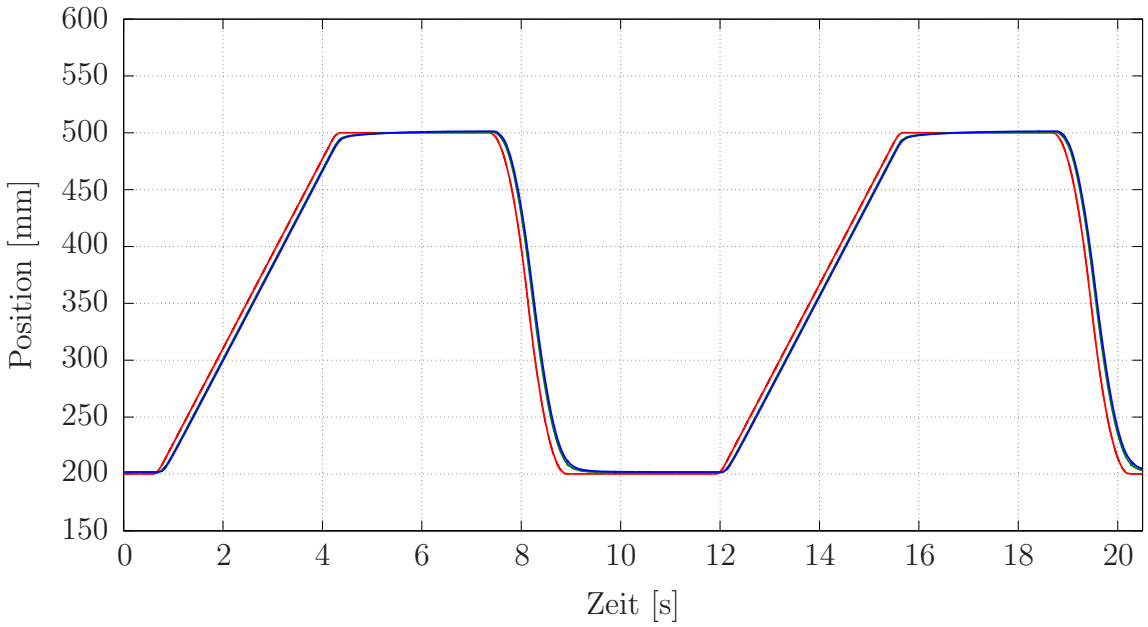

Abbildung 4.7: Verhalten des realen Prüfstandes bei Verwendung des modellbasiert entwickelten Gleichlaufreglers (rot: Sollposition beider Zylinderkolben; grün: Istposition Zylinder Achse 1; blau: Istposition Zylinder Achse 2)

virtuellen Inbetriebnahme kann lediglich der auf der Steuerung vorhandene Reglercode getestet werden, sodass beispielsweise keine Fehler bei der Montage oder der Inbetriebnahme der Elektrik, z.B. durch fehlerhafte Verkabelung im Schaltschrank, erkannt werden können. Diese finale Inbetriebnahme an der realen Anlage benötigt nach vorheriger Durchführung einer virtuellen Inbetriebnahme jedoch weniger Zeit, da ein Teil des Inbetriebnahmeaufwands (z.B. das Finden und Beheben von Softwarefehlern sowie die Validierung des Reglercodes) bereits parallel zur Fertigung der Anlage angefallen ist.

In Abbildung 4.6 ist der reale Aufbau des Gleichlaufprüfstandes zu sehen, der im Rahmen dieses Beispiels verwendet wird. Dieser besteht aus den beiden mechanisch unabhängigen Zylindern links und rechts, jeweils einem Ventil auf der Rückseite der Anordnung sowie einer Druckversorgung des Systems. Die verwendete Industriesteuerung des Typs IndraControl XM22 sowie benötigte I/O-Module zur Anbindung der Sensoren an die Steuerungshardware befinden sich im vorderen Bereich der Abbildung. Das Verhalten der realen Anlage bei Verwendung des mit Hilfe der modellbasierten Entwicklungsmethoden entwickelten Gleichlaufreglers ist in Abbildung 4.7 dargestellt. Es ist zu erkennen, dass die Funktionalität des Reglers bereits ohne Anpassung der Regelparameter gegeben ist. Aufgrund der Abweichung zwischen Modell und realer Anlage wird die Güte des Ergebnisses der Simulation jedoch nicht ganz erreicht. An dieser Stelle ist die bereits zuvor erwähnte Feinanpassung der Parameter notwendig, um das Regelergebnis an der realen Anlage noch ein we-

nig zu verbessern. Die durch die Optimierung bestimmten Parameter stellen jedoch eine sehr gute Ausgangsbasis für die Feinanpassung dar.

4.1.5 Bewertung des modellbasierten Entwicklungsprozesses

Anhand des gezeigten Beispiels der Entwicklung eines Gleichlaufreglers lassen sich die Vorteile des modellbasierten Entwicklungsprozesses anschaulich verdeutlichen. Der im Rahmen dieser Arbeit präsentierte Entwicklungsprozess wird durchgehend von Modellen auf Basis offener Standards (durch Verwendung der Modellierungssprache Modelica) begleitet. Die vorgestellte Lösung steht damit sämtlichen Anwendern, insbesondere auch kleinen und mittleren Unternehmen, uneingeschränkt zur Verfügung. Durch einen Einsatz von Optimierungsverfahren zur Unterstützung der Auslegung wird erreicht, dass eine optimale Kombination von Komponenten und Reglerparametern bestimmt wird. Der Vorteil der Vorgehensweise ist, dass die erzielte Lösung von der Expertise des Ingenieurs unabhängig ist und auf diese Weise eine gleichbleibende, hohe Qualität erreicht wird.

Durch den modellbasierten Entwicklungsprozess können weiterhin bisher notwendige Re-Implementierungen vorhandenen Wissens vermieden werden. Dies zeigt sich besonders deutlich bei der Erstellung der Steuerungsapplikation, bei der der bereits in der Simulationsumgebung vorhandene Regler mit Hilfe der Methode der Codegenerierung in diese integriert wird, anstatt den Algorithmus in SPS-Programmiersprachen nach IEC 61131-3 zu reimplementieren. Darüber hinaus wurde auf Basis des zu Beginn des Entwicklungsprozesses erstellten Simulationsmodells der Regelstrecke eine virtuelle Inbetriebnahme durchgeführt. Diese ermöglicht es, bevor die reale Anlage existiert, den Steuerungscode in einer sicheren Umgebung zu testen. Sobald die Anlage errichtet ist, ist der Steuerungscode bereits validiert und getestet. Die abschließende Inbetriebnahme an der realen Anlage wird dadurch deutlich verkürzt, es ist in der Regel durch eine Abweichung im Verhalten zwischen Simulationsmodell und realer Anlage lediglich eine Feinanpassung der Parameter notwendig.

Sowohl die Codegenerierung zur Einbindung von Simulationsmodellen in die Steuerungsapplikation als auch die virtuelle Inbetriebnahme sind ohne Modifikationen mit der originalen Steuerungshardware verwendbar, die schließlich an der realen Anlage verwendet wird. Es wird beispielsweise keine spezielle Steuerungsfirmware benötigt. Gleichzeitig werden weiterhin die gewohnten Entwicklungstools für die Steuerungsapplikation verwendet, eine Einarbeitung in neue Softwarewerkzeuge ist nicht erforderlich. Dies unterscheidet den im Rahmen dieser Arbeit beschriebenen Ansatz deutlich von den bisher zur Verfügung stehenden Möglichkeiten, für die entweder spezielle Softwareumgebungen notwendig sind oder besondere Echtzeithardware benötigt wird, die letztlich nicht an der realen Anlage verwendet wird, sodass auch in diesen Fällen eine Überführung des Codes von der Erprobungshardware auf die schließlich an der Anlage verwendete Steuerungshardware notwendig ist, was wiederum dem Grundsatz der durchgängigen Entwicklung widerspricht.

4.2 Modellbasierte Entwicklung einer Flaschenabfüllanlage

Im vorherigen Beispiel wurde gezeigt, wie die im Rahmen dieser Arbeit vorgestellten modellbasierten Entwicklungsmethoden eingesetzt werden können, um eine effiziente Auslegung und Inbetriebnahme einer neuen technischen Anlage zu realisieren. Dabei basierte der gesamte Entwicklungsprozess auf der Verwendung offener Standards sowie Open Source Software. In dem in diesem Abschnitt dargestellten zweiten Beispiel wird stattdessen mit der *3DEXPERIENCE Plattform* (3DXP) der Firma Dassault Systèmes eine verbreitete, kommerzielle Softwarelösung für die Umsetzung einer modellbasierten Produktentwicklung verwendet. Wie in Abschnitt 3.1.1 beschrieben, unterstützt dieses Werkzeug die Modellierung in der offenen Sprache Modelica und deckt die Entwicklungsphasen von der Festlegung der Anforderungen bis zur simulativen Auslegung der Anlage ab. Eine Überführung der Modelle von der Phase der Auslegung in die nachfolgenden Phasen der Inbetriebnahme und des Betriebs sowie die Möglichkeit einer virtuellen Inbetriebnahme erlaubt das Tool standardmäßig nicht. Insbesondere an dieser Stelle lassen sich jedoch erhebliche Effizienzsteigerungen erzielen.

Zur Realisierung dieser Effizienzsteigerungen ist es wünschenswert, die im Rahmen dieser Arbeit entwickelten Werkzeuge, insbesondere die Toolchain zur automatischen Generierung von ausführbarem Reglercode sowie die Schnittstellen zur Reali-

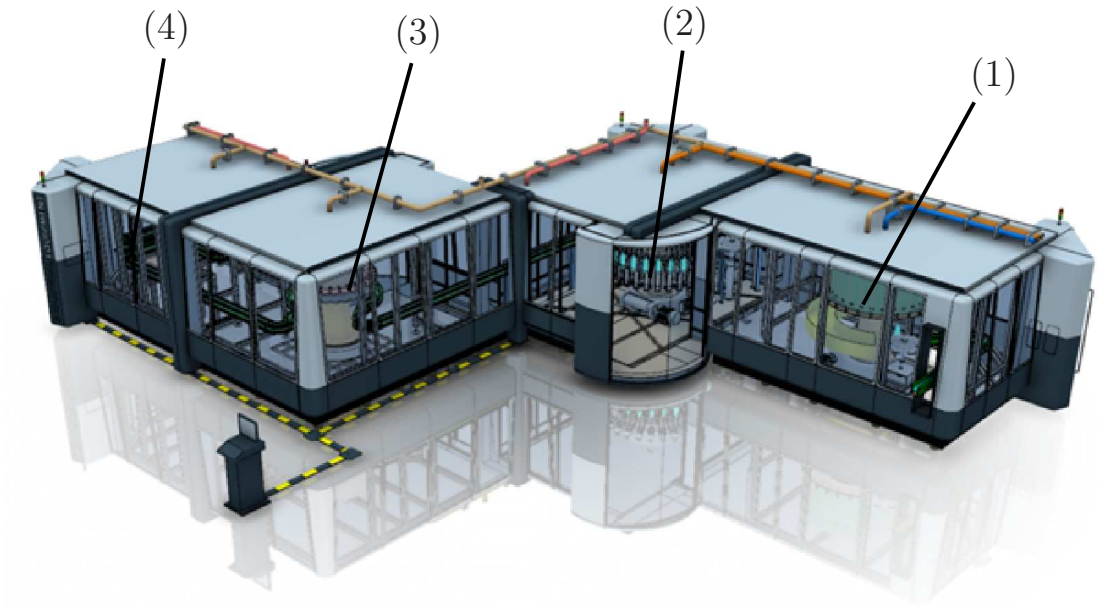

Abbildung 4.8: Gesamtbild der Anlage: (1) Spülen der Flaschen; (2) Befüllen der Flaschen; (3) Verschließen der Flaschen; (4) Etikettieren der Flaschen

sierung einer virtuellen Inbetriebnahme innerhalb der 3DXP verfügbar zu machen. An dieser Stelle wird besonders der Vorteil der offenen Schnittstellen deutlich. Die entwickelten Werkzeuge basieren vollständig auf offenen Standards, wodurch eine Integration auch in kommerzielle Tools auf einfache Weise ermöglicht wird. Dies wird im Folgenden am Beispiel der Entwicklung einer Flaschenabfüllanlage verdeutlicht. Eine derartige Anlage besteht in der Regel aus mehreren Modulen, die für die verschiedenen Prozessschritte verantwortlich sind. Die betrachtete Maschine besteht aus vier Modulen. Dabei sorgt je ein Modul für das Spülen der Flaschen, das Befüllen der Flaschen, das Verschließen der Flaschen sowie das Etikettieren der Flaschen. Die Maschine ist in Abbildung 4.8 dargestellt. Innerhalb dieses Beispiels wird die Methodik exemplarisch anhand des Moduls der Flaschenbefüllung veranschaulicht. Da die Methoden von allgemeiner Natur sind, können sie auf die anderen Module analog angewendet werden. Das vorgestellte Beispiel ist ebenfalls in [Hofmann *et al.* (2015a)] beschrieben.

4.2.1 Auslegung des Systems

Für die Entwicklung des Moduls zum Befüllen der Flaschen wird zunächst ein CAD-Modell erstellt. Die 3DXP unterstützt die Verwendung der Tools CATIA und SolidWorks. In diesem Beispiel wird für die Modellierung die Umgebung CATIA eingesetzt, die ein effizientes, computergestütztes Konstruieren der Maschine erlaubt. Mit Hilfe des CAD-Modells ist es jedoch nicht möglich, das dynamische Verhalten der Anlage zu untersuchen. In diesem Zusammenhang ist vor allem das Antriebskonzept zur Bewegung der Maschine von Interesse, welches nur durch eine ganzheitliche dynamische Systemsimulation bewertet werden kann. Eine Besonderheit der 3DXP ist es, im Sinne einer Durchgängigkeit des Entwicklungsprozesses ein dynamisches Mechanikmodell in Modelica automatisch aus der CAD-Beschreibung zu generieren. Dieses ist schließlich lediglich um Modelle der elektrischen Komponenten zum Antrieb der Mechanik zu ergänzen. Der Aufbau des Maschinenmodells sowie die automatische Generierung des Mechanikmodells werden in Abschnitt 4.2.1.1 beschrieben. Für die Bewegung der Maschine wird schließlich ein Regler benötigt, der in Abschnitt 4.2.1.2 entwickelt wird. Die Erstellung des Reglers erfolgt unter Verwendung der Bibliothek *mlpi4Modelica*[1] unmittelbar in der Simulationsumgebung.

4.2.1.1 Erstellung des Anlagenmodells

Das Befüllungsmodul besteht im Wesentlichen aus drei Teilen. Im ersten Abschnitt, der Einführeinheit, werden die zu befüllenden Flaschen über ein Transportband in das Modul eingeführt. Von dort aus werden sie über einen Drehmechanismus an die eigentliche Befülleinheit übergeben. Nach Durchlaufen der Befülleinheit entnimmt

[1] Wie bereits in Abschnitt 3.3.3.1 beschrieben, erlaubt das *Motion Logic Programming Interface* die Verwendung von Steuerungsbefehlen, beispielsweise zur Motionsteuerung von Achsen, unmittelbar aus Simulationsumgebungen heraus. Für die Verwendung innerhalb der Sprache Modelica steht die Modelica-Bibliothek mlpi4Modelica bereit.

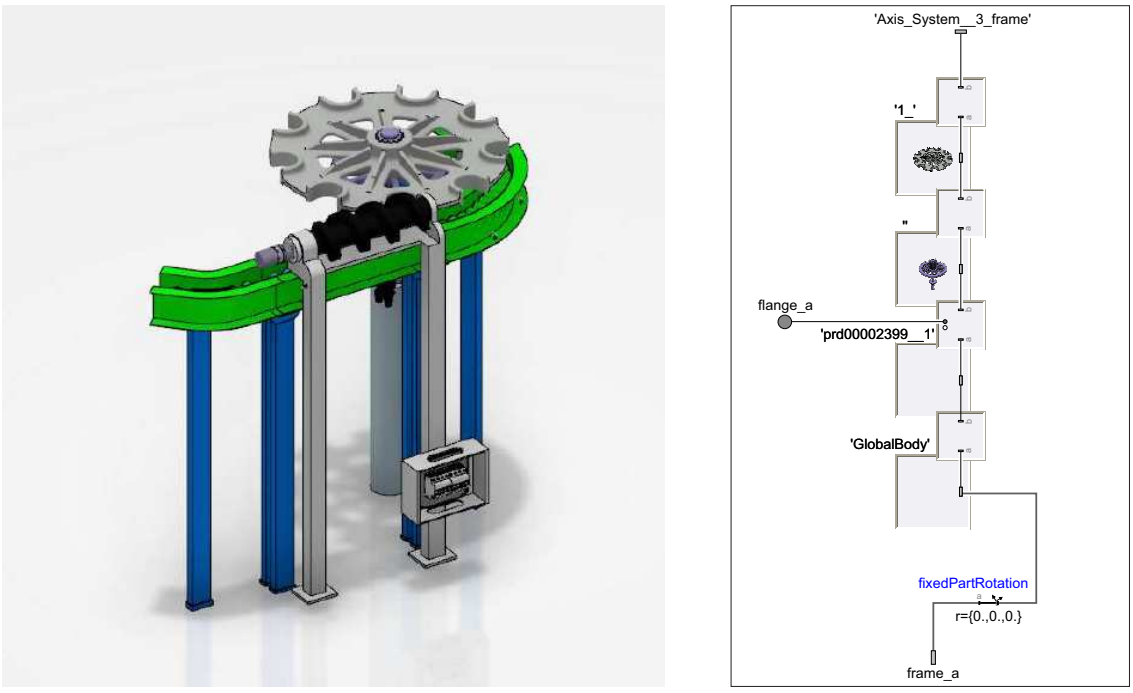

Abbildung 4.9: CAD-Modell der Einführeinheit (links); Modelica-Modell der Einführeinheit (rechts)

ein weiterer Drehmechanismus die Flaschen, sodass diese über ein Transportband aus dem Modul abgeführt werden können (Ausführeinheit). Exemplarisch ist das CAD-Modell der Einführeinheit in Abbildung 4.9 links dargestellt.

Neben der mechanischen Konstruktion ist die Auslegung des Antriebskonzeptes zur Sicherstellung der geforderten Dynamik von besonderer Bedeutung. In dem hier betrachteten Modul zum Befüllen der Flaschen befinden sich, wie zuvor beschrieben, drei Einheiten, die unabhängig voneinander angetrieben werden können. Für deren Antrieb sind jeweils geeignete Elektromotoren auszuwählen, die ausreichend Leistung für die Bewegung der Mechanik besitzen. Für ein tieferes Verständnis des dynamischen Verhaltens der Anlage reicht die Verwendung eines CAD-Modells nicht aus. Stattdessen ist es notwendig, eine dynamische Systemsimulation durchzuführen, bei der physikalische Modelle der Elektromotoren mit der Mechanik gekoppelt werden. Da die Mechanik jedoch lediglich in Form eines CAD-Modells vorliegt, welches für die dynamische Systemsimulation nicht unmittelbar verwendet werden kann, ist es bisher zumeist notwendig, das dynamische Mechanikmodell von Grund auf neu zu erstellen. Dieses Vorgehen ist jedoch äußerst ineffizient, da das CAD-Modell bereits alle relevanten Informationen wie Trägheiten, Abmaße und Anordnung der Körper und Gelenke enthält. Im Rahmen eines durchgängigen Engineerings ist es wünschenswert, das dynamische Mechanikmodell für die Systemsimulation automatisiert aus dem CAD-Modell zu erzeugen, sodass eine Re-Implementierung bereits im CAD-Modell enthaltenen Wissens vermieden werden kann. Die 3DXP

stellt Funktionalitäten zur Verfügung, aus einem CAD-Modell in CATIA automatisiert ein Modelica-Modell auf Basis der Modelica-Bibliothek *CATIA MultiBody* zu generieren [Baumgartner & Pfeiffer (2014)]. Angetriebene Gelenke innerhalb der CAD-Darstellung werden dabei automatisch in entsprechende Gelenke innerhalb des Modelica-Modells transformiert, die den Anschluss eines Motormodells erlauben. Das automatisiert generierte Modelica-Modell der Einführeinheit ist in Abbildung 4.9 rechts dargestellt.

Für den Antrieb der Mechanik werden *IndraDyn S* Synchron-Servomotoren der MSK-Baureihe [Bosch Rexroth AG (2014)] sowie Umrichter des Typs *IndraDrive HCS* verwendet. Die im Rahmen dieser Arbeit verwendeten Modelica-Modelle der Komponenten entstammen einer Rexroth-eigenen Modelica-Bibliothek und sind be-

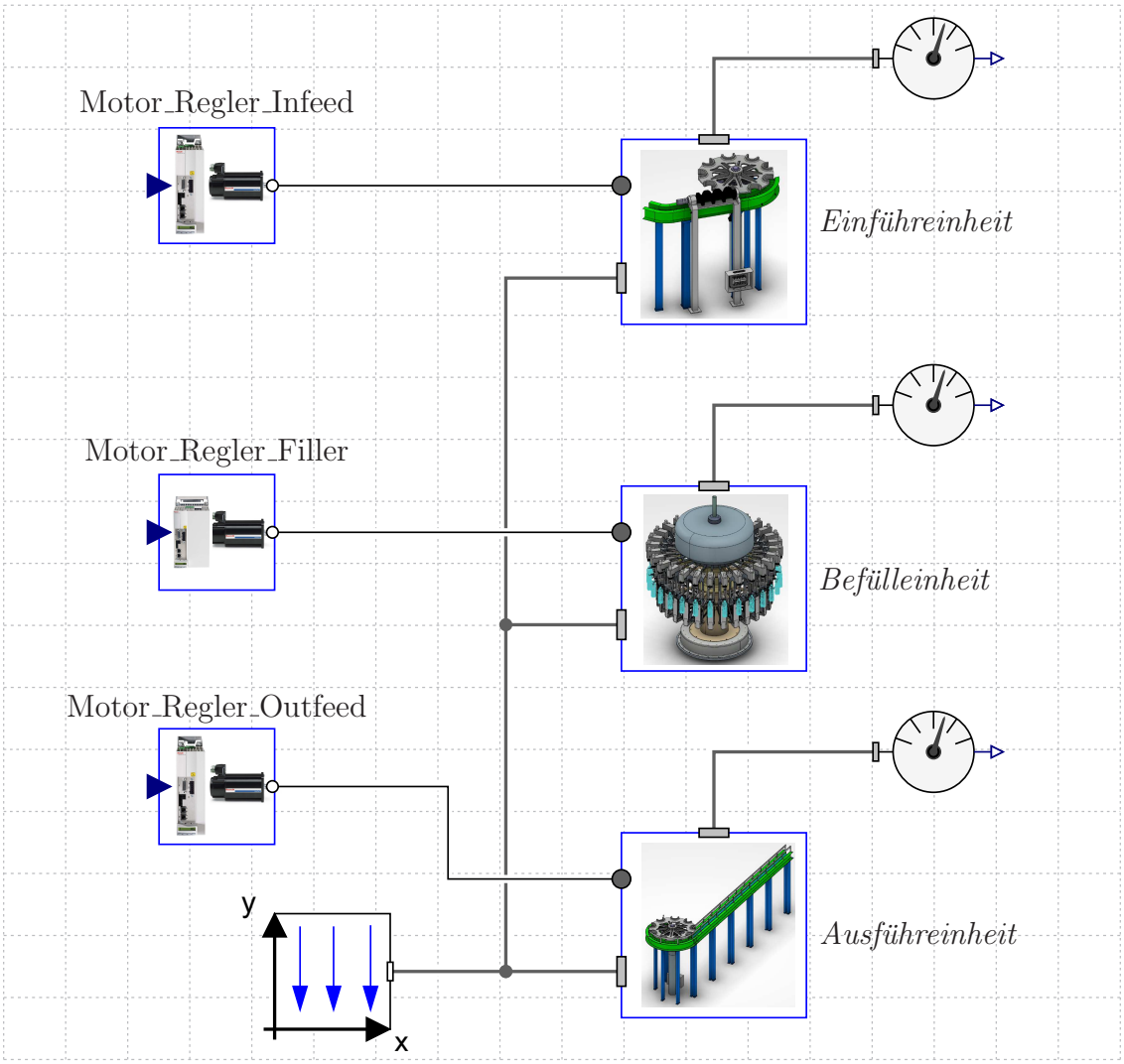

Abbildung 4.10: Dynamisches Modell des Befüllungsmoduls, bestehend aus dem Mechanikmodell und den Motor- bzw. Umrichtermodellen

reits vorparametriert, d.h. sie enthalten die komponentenspezifischen Kennlinien, beispielsweise für die Drehmoment- und Drehzahlbegrenzung. Das sich ergebende Gesamtmodell ist in Abbildung 4.10 dargestellt. Es ist zu beachten, dass dieses Systemmodell zum einen aus dem automatisch generierten Mechanikmodell und zum anderen aus bereits vorkonfigurierten Modellen der Motoren und Umrichter besteht, sodass dieses Gesamtmodell praktisch ohne Zusatzaufwand aus den vorherigen Entwicklungsphasen abgeleitet werden kann.

Wie in dem Modell in Abbildung 4.10 zu erkennen ist, sind die Eingänge der Antriebsmodelle noch nicht verbunden. Das jeweils anzuschließende Signal beschreibt das Sollverhalten des Antriebs und entstammt später dem Regelungsalgorithmus. In diesem werden die Sollpositionen bzw. die Sollgeschwindigkeiten der einzelnen Motoren berechnet. Der IndraDrive HCS, der gleichzeitig den Motorregler enthält, bestimmt aus der aktuellen Istposition des Motors und der gewünschten Sollposition den benötigten Strom, der als Eingang für den jeweiligen Motor dient. Die Entwicklung des Regelalgorithmus erfolgt im nächsten Abschnitt.

4.2.1.2 Entwicklung des Reglers

Die Hauptaufgabe der Regelung besteht darin, die drei unabhängigen Achsen des Moduls derart zu bewegen, dass sich eine synchronisierte Bewegung ergibt. Dies ist zwingend erforderlich, um die Flaschen von der Einführeinheit an die Befülleinheit und weiter an die Ausführeinheit zu übergeben. Für die Bewegung werden zwei unterschiedliche Betriebsmodi implementiert. In einem ersten Modus (*Manueller Modus*) können sich die drei Achsen zu Testzwecken unabhängig voneinander mit einer jeweils vorgegebenen Geschwindigkeit drehen, sodass in diesem Fall keine synchronisierte Bewegung notwendig ist. In dem zweiten Modus (*Automatik Modus*) ist die synchronisierte Bewegung erforderlich. Als Leitachse wird in diesem Fall die Bewegung der Befülleinheit festgelegt, für die eine Sollgeschwindigkeit vorgegeben wird. Sowohl die Einführeinheit als auch die Ausführeinheit synchronisieren sich zu dieser Bewegung.

Die Kommandierung der Achsen erfolgt unmittelbar unter Verwendung von Befehlen der *PLCopen Motion Control* Bibliotheken. Dies wird durch die Verwendung des *Motion Logic Programming Interface* ermöglicht. Für die Verwendung innerhalb eines Modelica-Modells steht innerhalb des Motion Logic Programming Interface die Bibliothek *mlpi4Modelica* bereit. Die Regelung wird mit Hilfe einer Ablaufsteuerung realisiert, die aus den fünf Zuständen *Initialisierung*, *Manueller Modus*, *Synchronisierung*, *Automatischer Modus* und *Stop* besteht. Zur Modellierung des Reglers wird die Modelica State Graph Bibliothek verwendet [Elmqvist *et al.* (2012)]. Das Modell des Reglers ist in Abbildung 4.11 dargestellt. Der Wechsel zwischen den Zuständen wird durch die Aktivierung entsprechender Transitionen ausgelöst. Die verschiedenen Zustände sind nachfolgend beschrieben.

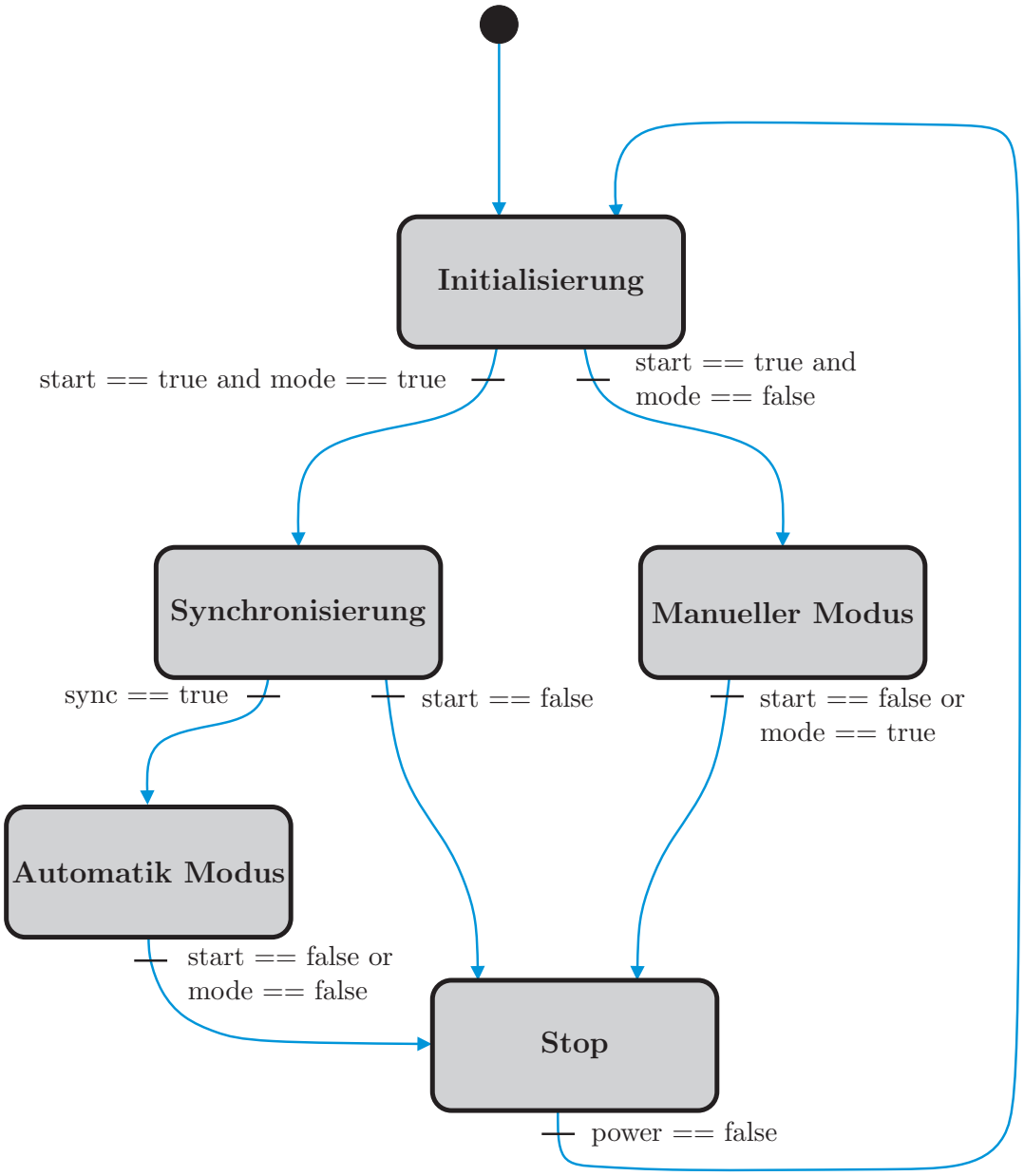

Abbildung 4.11: Struktur der Ablaufsteuerung zur Regelung des Befüllmoduls

- **Initialisierung**: Während des Hochlaufs befindet sich die Maschine zunächst in diesem Zustand. Sämtliche Achsen sind ausgeschaltet, sodass sich die Maschine nicht bewegt. Aus diesem Zustand kann die Maschine entweder in den Manuellen Modus (start == true and mode == true) oder in den Zustand Synchronisierung (start == true and mode == false) übergehen.

- **Manueller Modus**: Innerhalb dieses Zustandes bewegen sich die Achsen jeweils unabhängig voneinander. Für jede der drei Achsen stehen eine Variable zum Aktivieren bzw. Deaktivieren der Achse sowie entsprechende Variablen

für die Angabe der Geschwindigkeiten bereit. Dieser Modus ist insbesondere für Testfahrten geeignet, beispielsweise zum Anfahren bestimmter Positionen sowie zur Ermittlung optimaler Drehgeschwindigkeiten der einzelnen Achsen.

- **Synchronisierung**: Dieser Zustand ist eine Vorstufe zu dem Automatik Modus und wird aufgerufen, sobald vom Initialisierungszustand in den Automatik Modus gewechselt werden soll. Nachdem die aktuelle Position der drei Achsen erfasst wurde, wird in diesem Modus jede Achse auf eine definierte Referenzposition gefahren. Nach Durchlaufen des Zustandes Synchronisierung befinden sich die Achsen in einem synchronen Zustand, bewegen sich jedoch nicht.

- **Automatik Modus**: Nach Durchlaufen der Synchronisierung (sync == true) werden die Achsen in diesem Zustand synchronisiert hochgefahren bis zur gewünschten Geschwindigkeit. Als Leitachse dient die Befülleinheit, die Geschwindigkeiten der Einführ- sowie der Ausführeinheit ergeben sich dadurch automatisch.

- **Stop**: Der Zustand Stop kann aus jedem anderen Zustand, in dem sich die Maschine bewegt, direkt aufgerufen werden (durch Setzen von start == false). Der Zustand Stop bremst alle Achsen auf Geschwindigkeit Null herunter und überführt die Maschine in einen sicheren Zustand. Während das Herunterbremsen bei einem Aufruf aus dem Automatik Modus heraus synchronisiert geschieht, um eventuell in der Maschine befindliche Flaschen nicht zu zerstören, werden die Achsen bei einem Aufruf aus dem Manuellen Modus heraus individuell heruntergefahren. Nachdem alle Achsen stehen (power == false), wechselt der Zustand automatisch in Initialisierung. Wird der Modus gewechselt, d.h. zwischen Manuellem und Automatik Modus umgeschaltet, erfolgt dieser Wechsel ebenfalls über den Zustand Stop.

Mit der Entwicklung des Regelalgorithmus endet die Phase der Auslegung. Der nächste Schritt besteht darin, den Reglercode auf der realen Steuerungshardware, einer IndraControl XM22, zu implementieren und zu testen. Die 3DXP bietet an dieser Stelle jedoch standardmäßig keine Möglichkeit der Überführung des Wissens aus den Simulationsmodellen in die Phase der Inbetriebnahme. Dadurch ist ein durchgängiger, modellbasierter Entwicklungsprozess nicht umsetzbar, da der Reglercode in SPS Programmiersprachen innerhalb der Entwicklungsumgebung der Steuerung komplett neu implementiert werden müsste.

Im Folgenden wird daher gezeigt, wie die im Rahmen dieser Arbeit entwickelten Methoden der Codegenerierung und der virtuellen Inbetriebnahme zu einer Durchgängigkeit im Entwicklungsprozess beitragen können, auch wenn die ersten Schritte innerhalb einer kommerziellen Umgebung durchgeführt wurden.

4.2.2 Codegenerierung und virtuelle Inbetriebnahme

Da die verwendete Entwicklungsumgebung 3DXP auf offene Standards wie Modelica setzt und das Reglermodell daher in Modelica erstellt wurde, ist es nicht nur innerhalb der 3DXP verwendbar, sondern in allen Werkzeugen, die den Standard unterstützen. Insbesondere ist es möglich, die in Abschnitt 3.3 beschriebene Toolchain zur Generierung von ausführbarem Code aus dem Simulationsmodell des Reglers einzusetzen. Dadurch lässt sich das in der 3DXP modellierte Wissen für die Phasen der Inbetriebnahme und des Betriebs weiterverwenden, sodass ein durchgängiger Entwicklungsprozess auch über die 3DXP hinaus sichergestellt werden kann. Dies unterstreicht einmal mehr die Wichtigkeit der offenen Standards bei der Umsetzung modellbasierter Entwicklungsprozesse.

Codegenerierung Da der Regler vollständig innerhalb der Simulationsumgebung entwickelt wurde und durch die Verwendung der Bibliothek mlpi4Modelica als Teil des Motion Logic Programming Interfaces bereits die PLCopen Befehle zum Kommandieren der Achsen beinhaltet, ist lediglich eine leere Steuerungsapplikation mit drei Achsen innerhalb der Entwicklungsumgebung der Steuerung anzulegen. Anschließend wird das Modelica-Modell des Reglers an die im Rahmen dieser Arbeit entwickelte Toolchain zur automatischen Codegenerierung übergeben. Nach Durchlaufen der Toolchain liegt der Code in Form einer kompilierten Bibliothek vor. Für die Integration der Bibliothek in die Steuerungsapplikation wird automatisch ein Funktionsbaustein erstellt, der in die Steuerungsapplikation importiert werden kann.

Virtuelle Inbetriebnahme Für eine frühzeitige Erprobung und Optimierung des Reglers in einer sicheren Umgebung ist es sinnvoll, diesen zunächst mit dem virtuellen Abbild der Regelstrecke zu koppeln. Parallel dazu kann mit der Fertigung der realen Anlage begonnen werden. Für die Einbindung realer Steuerungen in die Simulation wird der in Abschnitt 3.4.2.3 beschriebene HiL-Coupler verwendet. Diese Komponente ist in Modelica verfügbar und lässt sich damit problemlos in das in Abbildung 4.10 dargestellte Simulationsmodell der Regelstrecke innerhalb der 3DXP integrieren. Durch die in Abschnitt 3.4.2.2 beschriebene Methode zur Synchronisierung der Simulationsumgebung und der Steuerungsapplikation, die innerhalb des HiL-Couplers standardmäßig implementiert ist, kann das bereits während der Auslegungsphase erstellte Simulationsmodell des Befüllungsmoduls ohne Modifikationen verwendet werden. Insbesondere ist es nicht notwendig, dass das Modell in Echtzeit abläuft, was die im Rahmen dieser Arbeit entwickelte Lösung deutlich von bisher am Markt erhältlichen Lösungen unterscheidet. Auf diese Weise wird ein wesentlich höherer Grad an Durchgängigkeit erreicht.

Für die Übertragung der Signale zwischen Steuerung und Simulation wird die ebenfalls in Abschnitt 3.4.2.3 beschriebene I/O-Komponente verwendet. Der Regler benötigt, wie im vorherigen Abschnitt beschrieben, die aktuellen Positionen der

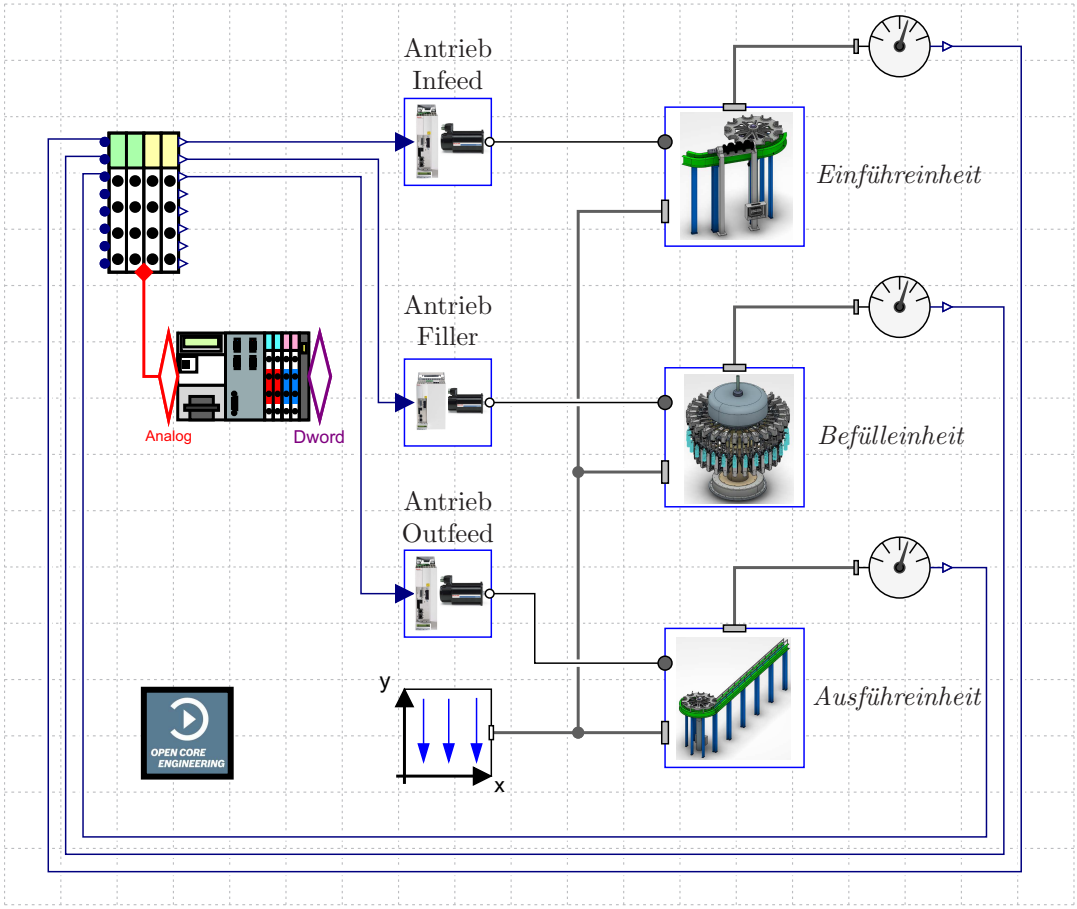

Abbildung 4.12: Gesamtmodell des Befüllungsmoduls mit HiL-Coupler Komponente für die Durchführung der virtuellen Inbetriebnahme

einzelnen Achsen als Eingang. Diese innerhalb der Simulation vorhandenen Signale werden mit den Eingängen des I/O-Blocks verbunden. Die Ausgänge des Reglers sind die Kommandowerte für die drei Achsen, die auf den jeweiligen Umrichter einwirken. Das für die virtuelle Inbetriebnahme verwendete Gesamtmodell ist in Abbildung 4.12 gezeigt.

Durch Starten der Simulation kann die virtuelle Inbetriebnahme durchgeführt werden. Als besondere Funktion innerhalb der 3DXP kann die Bewegung des Systems in der 3D-Ansicht auf Basis der CAD-Modelle verfolgt werden. Dies ist in Abbildung 4.13 dargestellt. Dadurch lassen sich die Maschinendynamik und die Synchronisierung der einzelnen Bauteile besonders anschaulich darstellen. Es ist zu beachten, dass diese Funktion zunächst eine spezielle Funktion der 3DXP ist. Es stehen jedoch auch unabhängig von der 3DXP Bibliotheken bereit, die eine 3D Visualisierung von mechanischen Modelica-Modellen ermöglichen. Neben Modelica-Bibliotheken, wie z.B. die *Visualization Library* des DLR[1] [Hellerer *et al.* (2014)], die die Vi-

[1]Deutsches Zentrum für Luft- und Raumfahrt

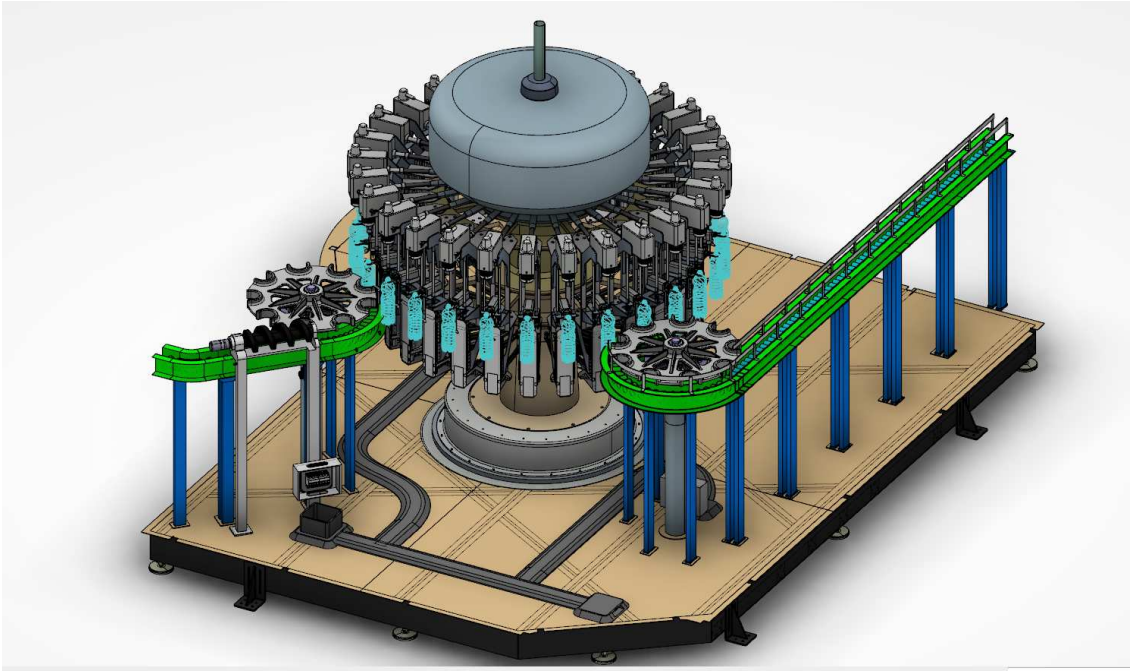

Abbildung 4.13: Visualisierung des Befüllungsmoduls auf Basis des CAD-Modells während der virtuellen Inbetriebnahme

sualisierung direkt in das Modelica-Modell einbetten, stehen zahlreiche 3D-Engines zur Verfügung, die parallel zur Simulation ausgeführt werden und mit den entsprechenden Daten aus der Simulation gespeist werden [Hofmann *et al.* (2015b)]. Die Parametrierung des Reglers kann online innerhalb der Entwicklungsumgebung der Steuerung verändert werden, während sich das veränderte Systemverhalten unmittelbar innerhalb der Simulation zeigt. Auf diese Weise ist es beispielsweise möglich, die maximalen Geschwindigkeiten bzw. Beschleunigungen der einzelnen Achsen optimal festzulegen. Durch die Verwendung physikalischer Motormodelle innerhalb der Simulation können dabei beispielsweise auftretende Schwingungen erkannt werden.

4.2.3 Bewertung und Fazit

Anhand des zuvor beschriebenen zweiten Beispiels, der Entwicklung einer komplexen Flaschenabfüllanlage, wurde gezeigt, wie die im Rahmen dieser Arbeit entwickelten Methoden zur Umsetzung modellbasierter Entwicklungsmethoden in der Praxis auch dann eingesetzt werden können, wenn bereits bestehende Software-Frameworks vorhanden sind. Auch in diesem Fall gelingt es, den Entwicklungsprozess durch eine Erhöhung des Grades der Durchgängigkeit effizienter zu gestalten.

Der Schlüssel für eine erfolgreiche Integration der in dieser Arbeit vorgestellten Methoden auch in kommerzielle Umgebungen sind offene Standards. Die in dem Beispiel verwendete *3DEXPERIENCE Plattform (3DXP)* unterstützt standardmäßig einen modellbasierten Entwicklungsprozess in den Phasen zwischen der Definition

der Anforderungen bis zur Auslegung des Systems. Eine Durchgängigkeit darüber hinaus, beispielsweise in die Phasen der Inbetriebnahme und des Betriebs, wird nicht unterstützt. Allerdings verwendet die 3DXP mit Modelica einen offenen Standard für die Modellbildung, wodurch die innerhalb der 3DXP erstellten Modelle mit den vorgestellten Methoden unmittelbar kompatibel sind. Im Wesentlichen ergeben sich durch die Verknüpfung der 3DXP und den im Rahmen dieser Arbeit vorgestellten Methoden drei konkrete Beiträge für eine Effizienzsteigerung, die im Anschluss erläutert werden:

- Ermöglichung einer effizienten Erstellung des Reglers unmittelbar in der Simulation durch Methoden des Motion Logic Programming Interface
- Automatische Erzeugung der Steuerungsapplikation aus dem Simulationsmodell des Reglers und damit einhergehende Ausdehnung der Durchgängigkeit hinein in die Phase der Inbetriebnahme und des Betriebs
- Ermöglichung einer virtuellen Inbetriebnahme als wesentliche Komponente innerhalb eines modellbasierten Entwicklungsprozesses auf Basis des vorhandenen Modells der Flaschenabfüllanlage sowie der HiL-Coupler Komponente

Eine erste Effizienzsteigerung, die sich durch die Integration der im Rahmen dieser Arbeit entwickelten Methoden in den Entwicklungsprozess ergibt, betrifft die Erstellung der Steuerungsapplikation. Die Verwendung der in Modelica verfügbaren Bibliothek *mlpi4Modelica* als Teil des Motion Logic Programming Interfaces (vgl. Abschnitt 3.3.3.1) ermöglicht, wie im vorherigen Abschnitt gezeigt wurde, die Erstellung der Steuerungsapplikation unmittelbar innerhalb der Simulationsumgebung, sodass auf eine Implementierung in SPS Programmiersprachen nach IEC 61131-3, wie sie bisher notwendig war, verzichtet werden kann. Innerhalb der Simulationsumgebung kann bei der Erstellung des Reglermodells auf Bibliotheken wie *State Graph* zur Beschreibung von Ablaufsteuerungen zurückgegriffen werden, was eine deutliche Vereinfachung gegenüber der Verwendung der PLCopen Befehle in SPS Programmiersprachen mit sich bringt. Durch die Tatsache, dass der Regelalgorithmus zunächst allgemein innerhalb der Simulationsumgebung verfügbar ist und nicht fest auf einer speziellen Hardwareplattform implementiert ist, lässt er sich anschließend auf einfache Weise auf verschiedene Steuerungen übertragen, wodurch eine Mehrfachverwendung deutlich vereinfacht wird. Ein weiterer Vorteil der Erstellung des Reglers innerhalb der Simulationsumgebung ist, dass auf diese Weise sämtliche Modelle innerhalb einer Plattform bereit stehen und nicht auf unterschiedliche Entwicklungswerkzeuge verteilt sind, was insbesondere für eine Archivierung des Projektes von Vorteil ist.

Des Weiteren gelingt es, durch die Verwendung der in Abschnitt 3.3 beschriebenen Toolchain die Modelle über die Phase der Auslegung hinaus zu verwenden, indem aus dem Simulationsmodell des Reglers automatisiert Code generiert wird, der auf

der Steuerungshardware ausgeführt wird. Die Toolchain ist unabhängig von einer Simulationsumgebung einsetzbar und basiert ganz allgemein auf Modelica-Modellen. Sie ist insbesondere unabhängig davon verwendbar, ob das Modell in einer freien oder einer kommerziellen Modelica-Umgebung erstellt wird. Da die Modellierung des Reglers innerhalb der 3DXP in Modelica erfolgt, kann das Reglermodell unmittelbar an die Toolchain übergeben werden, eine Anpassung ist nicht notwendig. Die Toolchain unterstützt weiterhin mit der IndraControl XM22 die später an der realen Anlage verwendete Steuerung, sodass sich eine Durchgängigkeit bis in den Betrieb hinein ergibt. Wird berücksichtigt, dass die 3DXP standardmäßig lediglich eine durchgängige Entwicklung bis zum Abschluss der Auslegungsphase unterstützt, ergibt sich durch die Einbindung der Toolchain in den Entwicklungsprozess eine deutliche Effizienzsteigerung durch die Vermeidung von Re-Implementierungen.

Wie ausführlich in Abschnitt 3.4 dargestellt, kann auch durch eine virtuelle Inbetriebnahme eine deutliche Effizienzsteigerung erreicht werden. Diese Möglichkeit steht innerhalb der 3DXP standardmäßig jedoch nicht zur Verfügung, sodass die in dieser Arbeit entwickelte Methode zur Umsetzung der virtuellen Inbetriebnahme mit der 3DXP kombiniert wurde. Wesentlicher Baustein zur Durchführung ist der in Abschnitt 3.4.2.3 beschriebene HiL-Coupler, der die Verbindung zwischen Steuerung und Simulation aufbaut und den synchronisierten Austausch der Daten gewährleistet. Diese Komponente ist als Modelica-Komponente verfügbar, sodass diese problemlos in das bestehende Modell der Regelstrecke integriert werden kann. An dieser Stelle wird erneut die Bedeutung der offenen Standards deutlich. Da Modelica eine Modellierungssprache und damit unabhängig von einem konkreten Simulationswerkzeug ist, können die Komponenten in jeder Umgebung, die den Modelica-Standard unterstützt, verwendet werden. Es spielt dabei keine Rolle, ob es sich um kommerzielle Werkzeuge oder um Umgebungen auf Basis von Open Source Software handelt. Durch die besondere Funktionsweise des HiL-Couplers, die eine Simulation der Regelstrecke in Echtzeit nicht erfordert, sondern stattdessen die Ausführung der Steuerungsapplikation an die Simulationsgeschwindigkeit anpasst, kann das Modell ohne Modifikation verwendet werden, wodurch ein maximaler Grad von Wiederverwendbarkeit und Durchgängigkeit erreicht wird. Zusätzlich lassen sich auf diese Weise die speziellen Funktionen der 3DXP nutzen, um beispielsweise eine Kopplung der dynamischen Simulation mit der 3D CAD-Ansicht zu ermöglichen. Dadurch lassen sich die Bewegungen der Anlage in der Simulation besonders anschaulich in der 3D-Ansicht analysieren.

KAPITEL 5

Zusammenfassung und Ausblick

Dieses abschließende Kapitel fasst zunächst die Arbeit zusammen. Dabei wird insbesondere überprüft, inwieweit die innerhalb der Einleitung formulierten Ziele erreicht sind. Zudem wird der wissenschaftliche Beitrag dieser Arbeit bewertet. Abschließend wird ein Ausblick auf nachfolgende Arbeiten gegeben.

5.1 Zusammenfassung

Die steigende Komplexität heutiger Produkte, die vor allem im Kontext von Industrie 4.0 immer häufiger cyber-physische Systeme darstellen, erfordert aufgrund der Tatsache, dass diese Produkte durch den stetig steigenden Wettbewerb auf den Märkten zusätzlich immer schneller entwickelt werden müssen, neuartige Entwicklungsprozesse. Die bisherige Entwicklungsgrundlage, die VDI Richtlinie 2206 zur Entwicklung mechatronischer Systeme, reicht für derartige Systeme nicht länger aus. Die Entwicklung cyber-physischer Systeme ist nur mit einem konsequenten Einsatz modellbasierter Entwicklungsmethoden zu bewältigen.

Nach bisherigem Stand der Technik werden modellbasierte Entwicklungsmethoden, bei denen der gesamte Entwicklungsprozess von den Anforderungen bis hin zum Betrieb der Anlage durch Simulationsmodelle begleitet wird, in der Praxis nur selten eingesetzt. Ein Grund dafür ist die Tatsache, dass kaum Werkzeuge am Markt erhältlich sind, die ein durchgängiges Engineering über den gesamten Entwicklungsprozess unterstützen. Weiterhin fallen für die Werkzeuge, die zumindest teilweise eine modellbasierte Entwicklung ermöglichen, hohe Lizenzkosten an, sodass ein Einsatz vor allem für kleine und mittlere Unternehmen nicht wirtschaftlich ist. Wesentliches Ziel dieser Arbeit war es, die Methoden der modellbasierten Entwicklung in

der Praxis umsetzbar und auch für kleine und mittlere Unternehmen zugänglich zu machen. Es ist zu beachten, dass die Umsetzung nicht auf Basis kommerzieller Tools möglich ist. Derartige Tools stellen zumeist abgeschlossene Systeme dar und erlauben somit eine durchgängige Entwicklung, die auf einer Weiterverwendung bereits modellierten Wissens in nachfolgenden Entwicklungsschritten basiert, nicht. Der Fokus dieser Arbeit liegt daher auf einer Lösung basierend auf offenen Standards, die hauptsächlich die Entwicklungsphasen der Auslegung sowie der Inbetriebnahme betrachtet.

Dazu wurden in Abschnitt 3.1.1 zunächst der gängige Entwicklungsprozess nach VDI Richtlinie 2206 vorgestellt und Schwachstellen bezüglich eines Einsatzes bei der Entwicklung aktueller Produkte aufgedeckt. Basierend darauf wurde in Abschnitt 3.1.3 ein durchgängiger, modellbasierter Entwicklungsprozess präsentiert. Gleichzeitig wurden Komponenten definiert, die für einen Einsatz der Methoden in der Praxis erforderlich sind, jedoch nach dem aktuellen Stand der Technik entweder nicht existieren oder lediglich Bestandteil kommerzieller Software-Frameworks sind, sodass die Lösungen nur ausgewählten Unternehmen zur Verfügung stehen. Diese Komponenten sind innerhalb dieser Arbeit entwickelt und umgesetzt worden. Zur Realisierung einer Lösung auf offenen Standards ist es insbesondere notwendig, auch die Modellbildung auf Basis einer offenen, standardisierten Modellierungssprache durchzuführen. Sämtliche Komponenten dieser Arbeit basieren auf Modellen der freien Modellierungssprache Modelica, die sich aufgrund des objektorientierten Modellierungsansatzes ausgezeichnet sowohl für die Modellbildung von Regelstrecken als auch zur Modellbildung von Regelalgorithmen eignet.

Zunächst wurde in Abschnitt 3.2 ein Optimierungswerkzeug für eine automatisierte Auslegung von Systemen präsentiert. Die im Rahmen einer automatisierten Auslegung auftretenden Optimierungsprobleme suchen in der Regel nach einer optimalen Kombination aus Systemkomponenten sowie Reglerparametern. Da es sich bei Systemkomponenten um diskrete Optimierungsvariablen handelt, führt die Aufgabenstellung auf ein gemischt-ganzzahliges Optimierungsproblem, welches spezielle Lösungsverfahren notwendig macht. Zur Lösung dieser Klasse von Problemen wurden ein speziell für gemischt-ganzzahlige Optimierungsaufgaben angepasster Genetischer Algorithmus sowie ein Partikelschwarmoptimierer in das Optimierungswerkzeug integriert. Bei der Modifikation des Genetischen Algorithmus wurde zusätzlich berücksichtigt, dass es sich bei den Optimierungsproblemen um die Optimierung von technischen Systemen handelt. Der Rekombinationsoperator ist in der Lage, auf Komponenteninformationen zurückzugreifen, um eine sinnvolle Rekombination zu ermöglichen. Eine derartige Möglichkeit steht bisher in gängigen Optimierungstools nicht zur Verfügung. Weiterhin wurde eine parallele Auswertung der einzelnen Individuen des Genetischen Algorithmus in das Optimierungswerkzeug integriert, um die benötigte Rechenzeit zu reduzieren.

Nachdem das System mit Hilfe von Optimierungsmethoden und unter Verwendung eines Simulationsmodells ausgelegt ist, wird eine Werkzeugkette benötigt, die es

ermöglicht, aus der Modellbeschreibung ausführbaren Code zu erzeugen, der in Echtzeit auf der an der Maschine verwendeten Steuerungshardware ausgeführt werden kann. Eine derartige Werkzeugkette wurde in Abschnitt 3.3 entwickelt und vorgestellt. Mit dieser Werkzeugkette ist es einerseits möglich, im Sinne eines Rapid Control Prototyping Code aus dem Reglermodell zu generieren. Andererseits kann zur Umsetzung beispielsweise einer modellbasierten Regelung oder einer modellbasierten Diagnose ebenso das Streckenmodell den Ausgangspunkt für die Codegenerierung darstellen. Für die Simulation von Modellen auf der Steuerung wird ein echtzeitfähiger Simulationskern benötigt, der die für die Simulation notwendigen numerischen Verfahren beinhaltet. Zusätzlich steuert der Simulationskern den Ablauf der Simulation und übernimmt die Behandlung von Unstetigkeiten und algebraischen Schleifen. Die Entwicklung eines echtzeitfähigen Simulationskerns wurde ausführlich in Abschnitt 3.3.2 behandelt.

Innerhalb des vorgestellten modellbasierten Entwicklungsprozesses ist es vorgesehen, die Steuerungsapplikation mit Hilfe einer virtuellen Inbetriebnahme an der virtuellen Regelstrecke zu testen, bevor die reale Anlage existiert. Die Einbindung der Steuerungshardware zur Durchführung einer Hardware-In-The-Loop-Simulation macht schließlich die Entwicklung von Schnittstellen zwischen Simulationsumgebung und Steuerungshardware notwendig. Diese Schnittstellen wurden im Abschnitt 3.4 entwickelt und behandelt. Die Durchführung einer virtuellen Inbetriebnahme, bei der das Streckenmodell in Echtzeit simuliert wird, stellt den naheliegendsten Fall dar. Diese Variante kann jedoch nicht in jedem Fall verwendet werden. Insbesondere bei Verwendung komplexer Streckenmodelle kann eine Simulation in Echtzeit zumeist nicht garantiert werden, weswegen eine Modellanpassung zur Reduzierung der Komplexität notwendig ist. Diese Modellanpassung widerspricht jedoch dem angestrebten Konzept der durchgängigen Entwicklung. Stattdessen ist es wünschenswert, die Modelle aus der Auslegungsphase unverändert weiterzuverwenden. Dies gelingt durch die in Abschnitt 3.4.2 entwickelte nicht-echtzeitfähige virtuelle Inbetriebnahme. Die notwendige Synchronisierung zwischen der Simulationsumgebung und dem ausgeführten Steuerungscode beruht auf einem getriggerten Modus der Steuerung, sodass die Ausführung des Steuerungscodes aus der Simulationsumgebung heraus gesteuert werden kann. Dieses Feature wird derzeit ebenfalls von keinem erhältlichen Tool am Markt ermöglicht, es wurde im Rahmen dieser Arbeit patentiert.

Die Funktionsweise der entwickelten Lösung in der Praxis wurde anschließend anhand zweier Beispiele, der Auslegung und Inbetriebnahme einer hydraulischen Presse sowie der Entwicklung einer Flaschenabfüllanlage, demonstriert.

5.2 Wissenschaftlicher Beitrag der Arbeit

Mit Hilfe des im Rahmen dieser Arbeit vorgestellten modellbasierten Entwicklungsprozesses sowie der genannten Komponenten ist es möglich, die Entwicklung von industriellen Anlagen durchgängig und auf Basis offener Standards durchzuführen.

Dadurch lässt sich, verglichen mit dem bisher standardmäßig verwendeten Entwicklungsprozess für mechatronische Systeme nach VDI Richtlinie 2206, eine deutliche Effizienzsteigerung erzielen. Da die zur Umsetzung des vorgestellten modellbasierten Entwicklungsprozesses benötigten Werkzeuge bisher am Markt nicht erhältlich waren, jedoch im Rahmen dieser Arbeit entwickelt wurden, stellt diese Arbeit einen signifikanten Beitrag zur Anwendung modellbasierter Entwicklungsmethoden in der Praxis dar. Mit den bereitgestellten Methoden auf Basis offener Standards können auch kleine und mittlere Unternehmen Zugang zu modellbasierten Entwicklungsmethoden, deren Bedeutung in Zukunft durch die steigende Komplexität der Produkte stetig weiter zunehmen wird, erlangen. Die vorgestellten Werkzeuge greifen unmittelbar ineinander und sind mit Fokus auf eine einfache und intuitive Bedienung entwickelt worden. Somit sind die Anwender in der Lage, durch die erzielten Effizienzsteigerungen, die hauptsächlich die Phasen der Auslegung und Inbetriebnahme betreffen, ihre Time-to-Market zu senken, wodurch sie auch in Zukunft konkurrenzfähig bleiben können. Da die Methoden auf offenen Standards basieren und damit toolunabhängig sind, lassen sie sich auf einfache Weise in bestehende Entwicklungslandschaften integrieren, auch wenn es sich dabei um kommerzielle Softwareumgebungen handelt (vgl. Abschnitt 4.2).

Die Vorteile, die sich aus der Verwendung modellbasierter Entwicklungsmethoden ergeben, gehen jedoch weit über den in den Beispielen in Abschnitt 4 vorgestellten Anwendungsfall des *Rapid Control Prototypings*, bei dem Teile der Steuerungsapplikation im Sinne einer Durchgängigkeit automatisiert aus Simulationsmodellen heraus generiert wird, hinaus. Neben der Ausführung von Reglermodellen auf der Steuerungshardware ist es ebenso denkbar, alternative Modelle, beispielsweise Streckenmodelle der geregelten Anlage, parallel zum Betrieb der Anlage auf der Hardware zu simulieren. In diesem Kontext eröffnen sich zahlreiche Anwendungsgebiete, von denen im Folgenden einige beispielhaft angesprochen werden:

- *Modellbasierte Regelung*: Durch eine Simulation von Streckenmodellen auf der Industriesteuerung parallel zum Betrieb können relevante Größen simulativ erfasst werden, die anschließend für eine Regelung verwendet werden. Beispielsweise ist es möglich, durch Verwendung eines Modells eines Roboters die an den Gelenken benötigten Momente der Motoren zur Ausführung einer bestimmten Bewegung zu ermitteln. Werden diese im Betrieb als Vorsteuerung für den Regelungsalgorithmus verwendet, kann die Regelungsgüte deutlich verbessert werden.

- *Modellbasierte Diagnose*: Durch die Verwendung von Simulationsmodellen parallel zum Betrieb der Anlage kann es gelingen, bevorstehende oder bereits aufgetretene Fehlerfälle zu detektieren. Die virtuelle Regelstrecke wird in jedem Schritt mit den aktuellen Eingängen der realen Maschine simuliert. Treten Abweichungen zwischen dem simulativ ermittelten Soll-Verhalten und dem tatsächlich an der Maschine gemessenen Signal auf, ist dies ein Anzeichen für einen Fehlerfall in der Anlage. Abhängig von der Güte der in den

Modellen vorhandenen Fehlermodelle lässt sich unter Umständen sogar auf den genauen Fehlerfall schließen. Die Verwendung von Simulationsmodellen parallel zum Betrieb erlaubt folglich die Umsetzung von *Predictive Maintenance* Anwendungen.

- *Modellprädiktive Regelung*: Bei der modellprädiktiven Regelung wird das Regelungsproblem als dynamisches Optimierungsproblem unter Berücksichtigung der Systemdynamik als Nebenbedingung aufgefasst. In diesem Fall wird in jedem Zeitschritt ein Optimalsteuerungsproblem auf der Steuerungshardware gelöst, aus dem der optimale Stelleingang auf das System bestimmt wird. Dadurch, dass das während der Auslegungsphase erstellte Streckenmodell bis in die Phase des Betriebs überführt und in dieser zur Regelung der Anlage verwendet wird, lässt sich das Konzept der durchgängigen, modellbasierten Entwicklung anhand der modellprädiktiven Regelung sehr gut zeigen.

Ganz allgemein wird mit Hilfe der modellbasierten Entwicklung und insbesondere der Verwendung von Modellen parallel zum Betrieb die Realisierung von *Smart Services* ermöglicht. Auch zur Umsetzung derartiger Methoden wurden in dieser Arbeit die Rahmenbedingungen geschaffen, da die vorgestellte Werkzeugkette zur Ausführung von Modellen auf Industriesteuerungen universell einsetzbar ist und damit nicht nur für Reglermodelle, sondern gleichzeitig für die Modelle der Regelstrecke verwendet werden kann. Das aus derartigen Modellen resultierende System gewöhnlicher Differentialgleichungen kann jedoch in der Regel mit den klassisch eingesetzten Methoden zur Echtzeitsimulation, wie beispielsweise dem Expliziten Euler-Verfahren, nicht gelöst werden. Im Rahmen dieser Arbeit wurde der verwendete Echtzeit-Simulationskern um entsprechende numerische Verfahren, die der Klasse der Rosenbrock-Verfahren entstammen, erweitert, sodass auch mathematisch steife Systeme in Echtzeit simuliert werden können. Dadurch ist im Rahmen dieser Arbeit ein Echtzeit-Simulationskern entstanden, der problemlos für die Implementierung von Smart Services angewendet werden kann. Die vorgestellten Methoden stellen daher ebenso einen Beitrag zur Umsetzung von Industrie 4.0 Anwendungen in der Praxis dar.

Zusätzlich stellt insbesondere die im Rahmen dieser Arbeit entwickelte und patentierte Methode, eine echtzeitfähige Steuerungshardware mit einer nicht-echtzeitfähigen Simulationsumgebung derart zu koppeln, dass die resultierende Hardware-In-The-Loop-Simulation stets synchron abläuft, eine Revolution dar. Damit ist es erstmals möglich, ein beliebig komplexes Streckenmodell zusammen mit einer Industriesteuerung für eine virtuelle Inbetriebnahme zu verwenden. Dies vereinfacht auf der einen Seite die virtuelle Inbetriebnahme enorm, da auf eine Anpassung des Modells zur Gewährleistung der Echtzeitfähigkeit verzichtet werden kann. Gleichzeitig kann dadurch die Qualität der virtuellen Inbetriebnahme deutlich gesteigert werden, da bei guter Modellbildung der Regelstrecke die Abweichung zwischen Modellverhalten und realer Maschine reduziert werden kann.

5.3 Ausblick

Mit den vorgestellten Werkzeuge ist es möglich, einen durchgängigen, modellbasierten Engineeringprozess in der Praxis umzusetzen, bei dem Informationen aus der Auslegungsphase bis hinein in den Betrieb weiterverwendet werden. Wie bereits erwähnt, liegt der Fokus innerhalb dieser Arbeit auf den Entwicklungsphasen der Auslegung und Inbetriebnahme. Jedoch lässt sich das Konzept des durchgängigen Engineerings sowohl auf vorgelagerte Schritte als auch auf nachgelagerte Schritte ausweiten.

Einbindung von Anforderungen in den Entwicklungsprozess Bei der Entwicklung eines neuen Produktes besteht der erste Schritt darin, die Anforderungen an das zu entwickelnde Produkt zu definieren. Für einen effizienten Entwicklungsprozess ist es sinnvoll, bereits die Anforderungen derart in Form eines Anforderungsmodells zu hinterlegen, dass diese während der nachfolgenden Entwicklungsphasen unmittelbar weiterverwendet können. Während der Entwicklung des Systemmodells können diese Informationen schließlich automatisch verarbeitet werden und, sofern es sich um funktionale Anforderungen handelt, überprüft werden. Besteht für eine hydraulische Anlage beispielsweise die Forderung, dass die Geschwindigkeit des Zylinderkolbens einen bestimmten Wert nicht überschreiten darf, so ist diese Information Bestandteil des Anforderungsmodells. Diese Informationen können bei der Erstellung des physikalischen Systemmodells automatisch in dieses integriert werden. Die Anforderung kann anschließend innerhalb der Simulation geprüft werden.

Für die Erstellung von Anforderungsmodellen existieren bereits Ansätze, wie in Abschnitt 3.1.1 dargestellt wurde. Für die Überprüfung der Anforderungen innerhalb des physikalischen Modelica-Modells steht beispielsweise eine Modelica-Bibliothek zur Verfügung [Otter *et al.* (2015)]. Um ein Anforderungsmodells zu erstellen, stehen mit SysML sowie weiteren UML-basierten Sprachen ebenfalls Möglichkeiten bereit. Die Einbindung derartiger Modelle in den durchgängigen Entwicklungsprozess sollte in weiteren Arbeiten ermöglicht werden.

Ableitung des Mechanikmodells aus einer CAD-Beschreibung Für die räumliche Konstruktion des zu entwickelnden Systems sowie der Untersuchung der Bauteile bezüglich mechanischer oder thermischer Last zur Ermittlung von Verformungen wird, nachdem die Anforderungen festgelegt sind, zumeist ein CAD-Modell erstellt. In diesem CAD-Modell sind sämtliche mechanischen Komponenten mit ihren Materialeigenschaften vorhanden. Bei dem Aufbau eines physikalischen Gesamtsystemmodells in Modelica müssen diese Informationen bisher in der Regel neu modelliert werden. Im Sinne eines durchgängigen Engineerings sollte es jedoch ermöglicht werden, die Komponenten inklusive der gespeicherten Informationen, wie beispielsweise Maße und Trägheitseigenschaften, automatisch zu überführen. In einigen kommerziellen Softwareframeworks, wie beispielsweise der 3DEXPERIENCE Plattform der Firma Dassault Systèmes oder MATLAB/Simulink bei Verwendung

des Simscape-Moduls, ist die Erzeugung von physikalischen Modellen aus CAD-Modellen möglich. In diesem Fall handelt es sich jedoch um kommerzielle Lösungen, die aufgrund hoher Lizenzkosten einer Vielzahl von kleinen und mittleren Unternehmen nicht zur Verfügung stehen. Zusätzlich sind diese Tools stets nur mit einigen wenigen CAD-Werkzeugen kompatibel. Um die Vorteile des modellbasierten Engineerings auch Anwendern kleiner und mittlerer Unternehmen zugänglich zu machen, sollte eine automatische Modellerzeugung aus CAD-Daten ebenfalls in den in dieser Arbeit vorgestellten Entwicklungsprozess integriert werden. Es ist zudem anzustreben, die Lösung nicht auf einzelne CAD-Werkzeuge zu beschränken. Die Entwicklung dieses Moduls auf Basis offener Schnittstellen ist bereits Bestandteil bestehender Arbeiten.

Umsetzung von Smart Services während des Betriebs Die in dieser Arbeit in Abschnitt 3.3 vorgestellte Werkzeugkette kann verwendet werden, um sowohl Modelica-Modelle von Reglern sowie Regelstrecken auf der Steuerungshardware in Echtzeit auszuführen. Neben den vorgestellten Möglichkeiten für den Einsatz der Modelle, wie beispielsweise das Rapid Control Prototyping oder die modellbasierte Regelung, sind eine Reihe weiterer Einsatzgebiete denkbar, die sich hauptsächlich durch die Vernetzung und Überwachung von Maschinen und Produktionssystemen im Kontext von Industrie 4.0 ergeben. Diese sogenannten Smart Services verwenden Maschinendaten und dienen dazu, die Produkte und Dienstleistungen innovativer und benutzerfreundlicher zu gestalten. So ist es beispielsweise möglich, im laufenden Betrieb aus den aufgenommenen Maschinendaten Wissen zu generieren, welches mit dem Maschinenhersteller geteilt wird. Auf Basis der Daten können anschließend Dienste zur Steigerung des Kundennutzens angeboten werden, z.B. eine vorausschauende Wartung. Für die Wissensgenerierung sind neben den Sensoren zur Aufnahme der Daten Modelle notwendig, die die Daten interpretieren. An dieser Stelle ist es denkbar, derartige Modelle mit Hilfe der Werkzeugkette auf der Steuerung auszuführen. Die Entwicklung von konkreten Umsetzungen für Smart Services ist ebenfalls Bestandteil aktueller Forschungsarbeiten.

ANHANG A

Anhang

A.1 Konfiguration des Optimierungstools

Das in Abschnitt 3.2 vorgestellte Optimierungsmodul innerhalb des OpenModelica-Frameworks wird, wie dort beschrieben, mit Hilfe einer Konfigurationsdatei parametriert. Diese Konfigurationsdatei ist eine Textdatei `OptimConfiguration.txt`, die sich in dem gleichen Ordner wie das Modelica-Modell befinden muss. Vor Beginn der Optimierung wird der Inhalt der Datei eingelesen und die dort angegebene Konfiguration verwendet. Neben einer in jedem Fall anzugebenden Basiskonfiguration, die beispielsweise das verwendete Optimierungsverfahren sowie die Dimension des Problems beinhaltet, gibt es eine verfahrensspezifische Konfiguration. Je nach verwendetem Optimierungsverfahren sind weiterhin Angaben zwingend notwendig. Ist der Parametername fettgedruckt dargestellt, ist die Konfiguration des Parameters erforderlich, es steht in diesem Fall kein Standardwert zur Verfügung. Parameter in normaler Schrift sind optional und müssen nicht angegeben werden. Ist kein Wert angegeben, wird der in der dritten Spalte genannte Standardwert verwendet. Die nachfolgend dargestellten Tabellen beinhalten sämtliche Parameter, die in der Konfigurationsdatei verwendet werden können. Für jeden Parameter ist eine neue Zeile in der Datei anzulegen, die Reihenfolge der Parameter ist beliebig.

Basiskonfiguration

Parameter	Erklärung	Standard
optim_mode	Gibt das verwendete Verfahren an: 1 = Partikelschwarmoptimierer 2 = Nelder-Mead-Verfahren 3 = Genetischer Algorithmus	-
dimension	Gibt die Dimension des Optimierungsproblems an (Anzahl der freien Optimierungsvariablen)	-
continuous	Gibt die Anzahl der reellwertigen Optimierungsvariablen an (continuous + discrete = dimension)	-
discrete	Gibt die Anzahl der ganzzahligen Optimierungsvariablen an (continuous + discrete = dimension)	-
optim_var	Enthält die Namen der freien Optimierungsvariablen, angefangen bei optim_var[0]. Die reellwertigen Optimierungsparameter müssen zu Beginn des Arrays stehen, die ganzzahligen am Ende	-
optim_obj	Enthält die Namen der Variablen für die Kostenfunktion, angefangen bei optim_obj[0]. Die Angabe von mehr als einer Kostenfunktion führt auf ein mehrkriterielles Optimierungsproblem, welches lediglich mit Hilfe des Genetischen Algorithmus gelöst werden kann	-

Tabelle A.1: Parameter der Basiskonfiguration

Verfahrensspezifische Konfiguration: Partikelschwarmoptimierung

Parameter	Erklärung	Standard
x_lo	Enthält die unteren Grenzen für die Optimierungsvariablen, angefangen bei x_lo[0]. Es muss für jede verwendete Optimierungsvariable eine untere Grenze angegeben werden	-
x_hi	Enthält die oberen Grenzen für die Optimierungsvariablen, angefangen bei x_hi[0]. Es muss für jede verwendete Optimierungsvariable eine obere Grenze angegeben werden	-
goal	Gibt den Kostenfunktionswert an, deren Unterschreitung zu einer Beendigung des Verfahrens führt (Genauigkeit erreicht)	0.1
size	Gibt die Anzahl an Individuen (Partikeln) innerhalb des Schwarmes an	30
print_every	Sofern dieser Wert größer als 0 ist, gibt er an, wie viele Schritte vergehen, bis die nächste Ausgabe auf dem Bildschirm erfolgt	0
steps	Enthält die maximale Anzahl an Schritten, die durchgeführt werden (Abbruchbedingung)	100000
c1	Gewichtung des Einflusses der kognitiven Komponente bei der Berechnung der neuen Geschwindigkeit des Partikels[1]	1.496[2]
c2	Gewichtung des Einflusses der sozialen Komponente bei der Berechnung der neuen Geschwindigkeit des Partikels[3]	1.496[4]
clamp_pos	1 = Anwendung strikter Randbedingungen: Überschreitet ein Partikel eine Randbedingung, wird die Position auf die Grenze festgesetzt und die Geschwindigkeit auf Null gesetzt 0 = Anwendung periodischer Randbedingungen	1

[1] vgl. Kapitel 2.3.2.2
[2] Standardwert nach [Clerc & Kennedy (2002)]
[3] vgl. Kapitel 2.3.2.2
[4] Standardwert nach [Clerc & Kennedy (2002)]

Parameter	Erklärung	Standard
nhood_strategy	Gibt die zu verwendende Nachbarschaftsstrategie an: 0 = Globale Nachbarschaftsstrategie 1 = Ringtopologie 2 = Zufallsbasierte Topologie	0
nhood_size	*Nur bei Verwendung einer zufallsbasierten Topologie:* Enthält die durchschnittliche Anzahl an Partikeln, mit denen ein Partikel in Kontakt steht	5
w_strategy	Gibt die zu verwendende Inertia Weight Strategie an: 0 = Konstanter Trägheits-Faktor; es wird der Wert w = 0.7298 verwendet[1] 1 = Linear abnehmender Trägheits-Faktor	0
w_min	*Nur bei Verwendung eines linear abnehmenden Trägheitsfaktors:* Enthält den minimalen Wert für den Trägheitsfaktor	0.3
w_max	*Nur bei Verwendung eines linear abnehmenden Trägheitsfaktors:* Enthält den maximalen Wert für den Trägheitsfaktor	0.7298

Tabelle A.2: Parameter der Konfiguration des Partikelschwarmoptimierers

[1] Standardwert nach [Clerc & Kennedy (2002)]

Verfahrensspezifische Konfiguration: Nelder-Mead-Verfahren

Parameter	Erklärung	Standard
start	Enthält die Startwerte für die reellwertigen Optimierungsvariablen, angefangen bei start[0].	-
step	Enthält die Schrittweite im ersten Schritt für die Optimierungsvariablen, angefangen bei step[0].	-
reqmin	Gibt die Änderung des Kostenfunktionswertes an, unterhalb der das Verfahren beendet wird (Genaugkeit erreicht)	1e-8
konvge	Gibt an, alle wie viele Iterationen eine Konvergenzüberprüfung durchgeführt wird	10
kcount	Enthält die maximale Anzahl an Funktionsauswertungen (Abbruchbedingung)	1000

Tabelle A.3: Parameter der Konfiguration des Nelder-Mead-Verfahrens

Verfahrensspezifische Konfiguration: Genetischer Algorithmus

Parameter	Erklärung	Standard
x_lo	Enthält die unteren Grenzen für die Optimierungsvariablen, angefangen bei x_lo[0]. Es muss für jede verwendete Optimierungsvariable eine untere Grenze angegeben werden	-
x_hi	Enthält die oberen Grenzen für die Optimierungsvariablen, angefangen bei x_hi[0]. Es muss für jede verwendete Optimierungsvariable eine obere Grenze angegeben werden	-
mu_nparents	Gibt die Anzahl der Eltern an	100
lambda_nchilds	Gibt die Anzahl der Nachkommen an	300
n_generations	Gibt die Anzahl der Generationen an	10
kappa	Gibt die Art des Selektions-Operators für die Population an 0 = Plus-Selektion $(\mu + \lambda)$[1] 1 = Komma-Selektion (μ, λ)[2]	0
s_init	Gibt den Startwert für die Mutationsschrittweite an	0.1
grid_divisions	*Nur bei Verwendung eines mehrkriteriellen Optimierungsproblems:* Gibt den Wert für die Unterteilung des Zielfunktionsraums für die Diversitäts-Wahrung an	32

Tabelle A.4: Parameter der Konfiguration des Genetischen Algorithmus

[1]Plus-Selektion: Aus den μ Eltern werden λ Nachkommen erzeugt ($\lambda > \mu$). Aus den Eltern und den Kindern werden die besten μ Individuen als Eltern für die nächste Generation ausgewählt. Es handelt sich um eine Elite-Selektion, bei der Individuen, die eine hohe Fitness haben, über mehrere Generation hinweg überleben.

[2]Komma-Selektion: Aus den μ Eltern werden λ Nachkommen erzeugt ($\lambda > \mu$). Es werden lediglich aus den Nachkommen die μ besten Individuen ausgewählt für die nächste Generation, die Eltern der Vorgängereltern werden nicht weiter berücksichtigt. Folglich lebt jedes Individuum genau eine Generation. Im Laufe der Generationen kann es passieren, dass das beste Individuum sich zwischendurch verschlechtert.

A.2 Parametrierung der Rosenbrock-Verfahren

$\gamma \quad = 7{,}886751345948129e-1$	
$a_{21} = 1{,}267949192431123e+0$ $a_{31} = 1{,}267949192431123e+0$ $a_{32} = 0{,}000000000000000e+0$	$c_{21} = -1{,}607695154586736e+0$ $c_{31} = -3{,}464101615137755e+0$ $c_{32} = -1{,}732050807568877e+0$
$\alpha_1 \quad = 0{,}000000000000000e+0$ $\alpha_2 \quad = 1{,}000000000000000e+0$ $\alpha_3 \quad = 1{,}000000000000000e+0$	$\gamma_1 \quad = \quad 7{,}886751345948129e-1$ $\gamma_2 \quad = -2{,}113248654051871e-1$ $\gamma_3 \quad = -1{,}077350269189626e+0$
$m_1 \quad = 2{,}000000000000000e+0$ $m_2 \quad = 5{,}773502691896258e-1$ $m_3 \quad = 4{,}226497308103742e-1$	

Tabelle A.5: Koeffizienten des ROS3P-Verfahrens [Lang & Verwer (2001)]

$\gamma \quad = 0{,}5728200000000000e+0$	
$a_{21} = 0{,}2000000000000000e+1$ $a_{31} = 0{,}1867943637803922e+1$ $a_{32} = 0{,}2344449711399156e+0$ $a_{41} = 0{,}1867943637803922e+1$ $a_{42} = 0{,}2344449711399156e+0$ $a_{43} = 0{,}0000000000000000e+0$	$c_{21} = -0{,}7137615036412310e+1$ $c_{31} = \quad 0{,}2580708087951457e+1$ $c_{32} = \quad 0{,}6515950076447975e+0$ $c_{41} = -0{,}2137148994382534e+1$ $c_{42} = -0{,}3214669691237626e+0$ $c_{43} = -0{,}6949742501781779e+0$
$\alpha_1 \quad = 0{,}0000000000000000e+0$ $\alpha_2 \quad = 0{,}1145640000000000e+1$ $\alpha_3 \quad = 0{,}6552168638155900e+0$ $\alpha_4 \quad = 0{,}6552168638155900e+0$	$\gamma_1 \quad = \quad 0{,}5728200000000000e-0$ $\gamma_2 \quad = -0{,}1769193891319233e+1$ $\gamma_3 \quad = \quad 0{,}7592633437920482e+0$ $\gamma_4 \quad = -0{,}1049021087100450e+0$
$m_1 \quad = 0{,}2255570073418735e+1$ $m_2 \quad = 0{,}2870493262186792e+0$ $m_3 \quad = 0{,}4353179431840180e+0$ $m_4 \quad = 0{,}1093502252409163e+1$	

Tabelle A.6: Koeffizienten des ROS4L-Verfahrens [Hairer & Wanner (1996)]

Literaturverzeichnis

Åkesson, J., Bergdahl, T., Gäfvert, M. & Tummescheit, H. (2009). Modeling and Optimization with Modelica and Optimica Using the JModelica.org Open Source Platform. In: *Proceedings of the 7th International Modelica Conference, Como, Italy, September 2009.*

Andersson, M. (1994). *Object-Oriented Modeling and Simulation of Hybrid Systems.* Ph.D. Thesis, Department of Automatic Control, Lund Institute of Technology (LTH).

Bauernhansl, T., Hompel, M. & Vogel-Heuser, B. (2014). *Industrie 4.0 in Produktion, Automatisierung und Logistik: Anwendung · Technologien · Migration.* Springer Fachmedien Wiesbaden.

Baumgartner, D. & Pfeiffer, A. (2014). Automated Modelica Package Generation of Parameterized Multibody Systems in CATIA. In: *Proceedings of the 10th International Modelica Conference, Lund, Sweden, March 2014.*

Beasley, D., Bull, D. R. & Martin, R. R. (1993). A Sequential Niche Technique for Multimodal Function Optimization. *Evolutionary Computation* **1**(2), 101–125.

Beine, M. (2009). Modellbasierte Entwicklung und Automatische Code-Generierung für sicherheitskritische Anwendungen. In: *7. Workshop Automotive Software Engineering (ASE 2009) im Rahmen der 39. Jahrestagung der Gesellschaft für Informatik (GI), Lübeck, September 2009.*

Bellalouna, F. (2009). *Integrationsplattform für eine interdisziplinäre Entwicklung mechatronischer Produkte.* Dissertation, Fakultät für Maschinenbau, Ruhr-Universität Bochum.

Bettenhausen, K. D. & Kowalewski, S. (2013). *Cyber-Physical Systems: Chancen und Nutzen aus Sicht der Automation.* Broschüre Thesen und Handlungsfelder, VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik.

Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauss, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T. *et al.* (2011). The Functional Mockup Interface for Tool Independent Exchange of Simulation Models. In: *Proceedings of the 8th International Modelica Conference, Dresden, Germany, March 2011.*

Blum, S. & Riedel, J. (2004). *Mehrzieloptimierung durch evolutionäre Algorithmen.* 1. Weimarer Optimierungs- und Stochastiktage, Dezember 2004.

Bosch Rexroth AG (2014). Projektierungsanleitung Rexroth IndraDyn S Synchronmotoren MSK (R911296288).

Bosch Rexroth AG (2015a). Betriebsanleitung Rexroth IndraControl VPB Steuerungen.

Bosch Rexroth AG (2015b). Betriebsanleitung Rexroth IndraControl XM2x Steuerungen (R911340666).

Brodtmann, T. & Malorny, C. (2014). *Zukunftsperspektive deutscher Maschinenbau - Erfolgreich in einem dynamischen Umfeld agieren.* Studie, durchgeführt von: Verband Deutscher Maschinen- und Anlagenbau e.V. sowie McKinsey & Company.

Brück, D., Elmqvist, H., Mattsson, S. E. & Olsson, H. (2002). Dymola for Multi-Engineering Modeling and Simulation. In: *Proceedings of the 2nd International Modelica Conference, Oberpfaffenhofen, Germany, March 2002.*

Buffoni, L. & Fritzson, P. (2014). Expressing Requirements in Modelica. In: *Proceedings of the 55th Scandinavian Conference on Simulation and Modeling (SIMS'2014), Aalborg, Denmark, October 2014.*

Burer, S. & Letchford, A. N. (2012). Non-Convex Mixed-Integer Nonlinear Programming: A Survey. *Surveys in Operations Research and Management Science* **17**(2), 97–106.

Cellier, F. E. & Kofman, E. (2006). *Continuous System Simulation.* Springer US.

Clerc, M. & Kennedy, J. (2002). The Particle Swarm-Explosion, Stability, and Convergence in a Multidimensional Complex Space. *IEEE Transactions on Evolutionary Computation* **6**(1), 58–73.

Dahl, O.-J., Myhrhaug, B. & Nygaard, K. (1968). *SIMULA 67 Common Base Language.* Norwegian Computing Center.

Dörner, D. (1994). Gedächtnis und Konstruieren. In: *Pahl, G. (Ed): Psychologische und pädagogische Fragen beim methodischen Konstruieren. Ergebnisse des Ladenburger Diskurses vom Mai 1992 - Oktober 1993. Köln: Verlag TÜV Rheinland, S. 150-160.*

Drogies, S. (2005). *Objektorientierte Modellbildung und Simulation des fahrdynamischen Verhaltens eines Kraftfahrzeuges*. Fortschritt-Berichte VDI. Reihe 12, Mensch-Maschine-Systeme.

Eberhart, R. C. & Kennedy, J. (1995). A New Optimizer using Particle Swarm Theory. In: *Proceedings of the 6th International Symposium on Micro Machine and Human Science, Nagoya, Japan, October 1995*.

Eitel, M. (2015). *Schwarmbasierte und evolutionäre Optimierungsverfahren für die Systemauslegung*. Masterarbeit, Hochschule Aschaffenburg.

Elmqvist, H. (1978). *A Structured Model Language for Large Continuous Systems*. Ph.D. Thesis, Department of Automatic Control, Lund Institute of Technology (LTH).

Elmqvist, H., Gaucher, F., Mattsson, S. E. & Dupont, F. (2012). State Machines in Modelica. In: *Proceedings of the 9th International Modelica Conference, Munich, Germany, September 2012*.

Emmerich, M. & Hosenberg, R. (2001). *TEA - A C++ Library for the Design of Evolutionary Algorithms*. Technischer Bericht CI106/01, Sonderforschungsbereich 531, Universität Dortmund.

Engels, E. & Gabler, T. (2012). Universelle Programmierschnittstelle für Motion-Logic Systeme. In: *Tagungsband zur AALE-Tagung 2012: 9. Fachkonferenz, Aachen, Mai 2012. Deutscher Industrieverlag, München*.

Esteez, E., Marcos, M. & Irisarri, E. (2009). Analysis of IEC 61131-3 Compliance through PLCopen XML interface. In: *Proceedings of the 7th IEEE International Conference on Industrial Informatics (INDIN), Cardiff, Wales, June 2009*.

Frenkel, J. (2014). *Entwicklung eines Modelica Compiler BackEnds für große Modelle*. Dissertation, Technische Universität Dresden, Fakultät für Maschinenwesen.

Fritzson, P., Aronsson, P., Bunus, P., Engelson, V., Saldamli, L., Johansson, H. & Karström, A. (2002). The Open Source Modelica Project. In: *Proceedings of the 2nd International Modelica Conference, Oberpfaffenhofen, Germany, March 2002*.

Fritzson, P., Aronsson, P., Lundvall, H., Nyström, K., Pop, A., Saldamli, L. & Broman, D. (2005). The OpenModelica Modeling, Simulation and Software Development Environment. *In Simulation News Europe* **44**.

Fritzson, P. & Engelson, V. (1998). Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In: *Proceedings of the 12th European Conference On Object-Oriented Programming (ECOOP), Brussels, Belgium, July 1998*. Springer.

Fritzson, P., Privitzer, P., Sjölund, M. & Pop, A. (2009). Towards a Text Generation Template Language for Modelica. In: *Proceedings of the 7th International Modelica Conference, Como, Italy, September 2009*.

Gausemeier, J. (2010). *Zuverlässigere Mechatronik – Forschungsergebnisse kompakt*. Broschüre, Transfer von Forschungsergebnissen aus 11 Verbundprojekten zur Steigerung der Zuverlässigkeit mechatronischer Systeme, herausgegeben vom Heinz Nixdorf Institut, Universität Paderborn.

Gausemeier, J., Dumitrescu, R. & Steffen, D. (2013). *Systems Engineering in der Praxis*. Studie, durchgeführt von: Heinz Nixdorf Institut der Universität Paderborn, Fraunhofer-Institut für Produktionstechnologie IPT - Projektgruppe Entwurfstechnik Mechatronik, UNITY AG.

Gausemeier, J. & Möhringer, S. (2003). Die neue Richtlinie VDI 2206: Entwicklungsmethodik für mechatronische Systeme. In: *VDI-Berichte 1753, VDI-Verlag, Düsseldorf*.

Gedda, S. (2011). *Calibration of Modelica Models using Derivative-Free Optimization*. Master's Thesis, Centre for Mathematical Sciences, Mathematics, Faculty of Engineering, Lund University.

Gehrke, M. (2005). *Entwurf mechatronischer Systeme auf Basis von Funktionshierarchien und Systemstrukturen*. Dissertation, Institut für Informatik, Universität Paderborn.

Gehrke, M., Meyer, J. & Schäfer, W. (2007). Modellierung von Softwarekomponenten für mechatronische Systeme in UML auf Basis von Systemstrukturen. In: *5. Paderborner Workshop: Entwurf mechatronischer Systeme, 22.-23.03.2007, Heinz Nixdorf Institut*.

Geiger, C. & Kanzow, C. (1999). *Numerische Verfahren zur Lösung unrestringierter Optimierungsaufgaben*. Springer Berlin Heidelberg.

Geissbauer, R., Schrauf, S., Koch, V. & Kuge, S. (2014). *Industrie 4.0 - Chancen und Herausforderungen der vierten industriellen Revolution*. Studie, durchgeführt von: PricewaterhouseCoopers Aktiengesellschaft - Wirtschaftsprüfungsgesellschaft, Oktober 2014.

Gill, P. E. & Wong, E. (2015). Methods for Convex and General Quadratic Programming. *Mathematical Programming Computation* **7**(1), 71–112.

Gottschling, J. & Schramm, D. (2014). *Numerische Methoden für Ingenieure*. 2. Auflage, Klartext Medienwerkstatt GmbH, Essen, Universität Duisburg-Essen.

Graichen, K. (2012). Methoden der Optimierung und optimalen Steuerung. Vorlesungsskript Wintersemester 2012/2013, Universität Ulm - Institut für Mess-, Regel- und Mikrotechnik.

Hairer, E., Nørsett, S. & Wanner, G. (1993). *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer Series in Computational Mathematics. Springer-Verlag Berlin Heidelberg.

Hairer, E. & Wanner, G. (1996). *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer Series in Computational Mathematics. Springer-Verlag Berlin Heidelberg.

Hellerer, M., Bellmann, T. & Schlegel, F. (2014). The DLR Visualization Library - Recent development and applications. In: *Proceedings of the 10th International Modelica Conference, Lund, Sweden, March 2014*.

Hirsch-Kauffmann, M. & Schweiger, M. (1992). *Biologie für Mediziner und Naturwissenschaftler*. Georg Thieme Verlag, Stuttgart.

Hofmann, A., Menager, N., Belhaj, I. & Mikelsons, L. (2015a). Integrated Engineering based on Modelica. In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 2015*.

Hofmann, A., Menager, N., Schweig, S. & Mikelsons, L. (2015b). Model-Based Engineering mit Industriesteuerungen. In: *8. Wissenschaftliche Fachtagung Verarbeitungsmaschinen und Verpackungstechnik (VVD), Dresden, März 2015*.

Hřebíček, J., Řezáč, M. *et al.* (2008). Modelling with Maple and MapleSim. In: *Proceedings of the 22nd European Conference on Modelling and Simulation (ECMS), Nicosia, Cyprus, June 2008*.

IEC (1993). *IEC Richtlinie 61131: Grundlagen Speicherprogrammierbarer Steuerungen*. Internationale Elektrotechnische Kommission (IEC).

Jackson, C. K. (2006). *Coordinating Engineering Disciplines*. The Mechatronics System Design Benchmark Report, by: AberdeenGroup, Inc., Boston.

Jeandel, A. & Boudaud, F. (1997). Physical System Modelling Languages: from ALLAN to Modelica. In: *Proceedings of the Building Simulation '97 - 5th International IBPSA Conference, Prague, Czech Republic, September 1997*.

Ji, H., Mikelsons, L., Kempf, K. & Schramm, D. (2012). Using Static Parametric Design to Support Systems Engineering of Industrial Automation Systems. In: *Proceedings of the 9th International Modelica Conference, Munich, Germany, September 2012*.

Kagermann, H., Wahlster, W. & Helbig, J. (2013). *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0.* Abschlussbericht des Arbeitskreises Industrie 4.0.

Kallrath, J. (2013). *Gemischt-ganzzahlige Optimierung: Modellierung in der Praxis: Mit Fallstudien aus Chemie, Energiewirtschaft, Metallgewerbe, Produktion und Logistik.* Vieweg+Teubner Verlag.

Kampfmann, R. (2014). *Echtzeitsimulation auf Industriesteuergeräten.* Masterarbeit, Technische Universität München.

Kara, I. B. (2015). *Design and Implementation of the ModelicaML Code Generator Using Acceleo 3.X.* Master's Thesis, Department of Computer and Information Science, Linköping University.

Kaufmann, T. (2015). *Geschäftsmodelle in Industrie 4.0 und dem Internet der Dinge: Der Weg vom Anspruch in die Wirklichkeit.* Springer Fachmedien Wiesbaden.

Kennedy, J. & Eberhart, R. (1997). A Discrete Binary Version of the Particle Swarm Algorithm. In: *IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation, Orlando, FL, USA, October 1997.*

Kirkpatrick, S. (1984). Optimization by Simulated Annealing: Quantitative Studies. *Journal of Statistical Physics* **34**(5), 975–986.

Kitayama, S., Arakawa, M. & Yamazaki, K. (2006). Penalty Function Approach for the Mixed Discrete Nonlinear Problems by Particle Swarm Optimization. *Structural and Multidisciplinary Optimization* **32**(3), 191–202.

Kramer, O. (2009). *Computational Intelligence: Eine Einführung.* Informatik im Fokus. Springer-Verlag Berlin Heidelberg.

Kreisselmeier, G. & Steinhauser, R. (1979). Systematische Auslegung von Reglern durch Optimierung eines vektoriellen Gütekriteriums. *at-Automatisierungstechnik* **27**(1-12), 76–79.

Lang, J. & Verwer, J. (2001). ROS3P — An Accurate Third-Order Rosenbrock Solver Designed for Parabolic Problems. *BIT Numerical Mathematics* **41**(4), 731–738.

Lazimy, R. (1982). Mixed-Integer Quadratic Programming. *Mathematical Programming* **22**(1), 332–349.

Lee, J. (2010). *Introduction to Topological Manifolds.* Graduate Texts in Mathematics, Vol. 940. Springer-Verlag New York.

Leute, U. (2004). *Physik und ihre Anwendungen in Technik und Umwelt*. Carl Hanser Verlag GmbH & Co. KG.

Leweling, U. (1995). *Möglichkeiten und Grenzen der modellbasierten Diagnose bei hydraulischen Anlagen*. Diplomarbeit, Universität-GH Paderborn, FB 17 Mathematik/Informatik.

Lindberg, R. (2010). *A Template-Based Code Generator for the OpenModelica Compiler*. Master's Thesis, Department of Computer and Information Science, Linköping University.

Matsui, T., Kato, K., Sakawa, M., Uno, T. & Matsumoto, K. (2008). Particle Swarm Optimization for Nonlinear Integer Programming Problems. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists, Hong Kong, March 2008*.

Mattsson, S. E. & Söderlind, G. (1993). Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives. *SIAM Journal on Scientific Computing* **14**(3), 677–692.

Melchior, J. (2008). *Implementierung von Partikel-Schwarm-Optimierung und Vergleich mit einer Evolutionsstrategie*. Bachelorarbeit, Fachbereich Informatik, Ruhr-Universität Bochum.

Menager, N. (2012). *Ein Ansatz zur automatisierten Auslegung von mechatronischen Systemen unter Verwendung einer objekt-orientierten Simulationsumgebung*. Masterarbeit, Fakultät für Ingenieurwissenschaften, Lehrstuhl für Mechatronik, Universität Duisburg-Essen.

Menager, N. (2013). An Approach for the Automated Design of Mechatronic Systems using an Object-Oriented Simulation Environment. In: *7th MODPROD Workshop on Model-Based Product Development, Linköping, Sweden, February 2013*.

Menager, N., Kampfmann, R., Worschech, N. & Mikelsons, L. (2015a). Suitability of Different Real-Time Solvers for a Model-Based Engineering Toolchain using Industrial Rexroth Controllers. In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 2015*.

Menager, N., Mikelsons, L. & Worschech, N. (2015b). Modellbasierte Entwicklung mit Rexroth Steuergeräten unter Nutzung von offenen Standards. In: *Tagungsband VDI Mechatroniktagung 2015, Dortmund, März 2015*.

Menager, N., Worschech, N. & Mikelsons, L. (2014a). A Toolchain for Rapid Control Prototyping using Rexroth Controllers and Open Source Software. In: *8th MODPROD Workshop on Model-Based Product Development, Linköping, Sweden, February 2014*.

Menager, N., Worschech, N. & Mikelsons, L. (2014b). A Toolchain for Rapid Control Prototyping using Rexroth Controllers and Open Source Software. In: *Proceedings of the 10th International Modelica Conference, Lund, Sweden, March 2014*.

Mewes, J. (2005). Virtuelle Inbetriebnahme mit realen Automatisierungssystemen und virtuellen Maschinen. In: *Deutsch-Niederländische Automatisierungstage 2005, Emden, Februar 2005*.

Mikelsons, L. (2012). *Generierung vereinfachter Modelle mechatronischer Systeme auf Basis symbolischer Gleichungen*. Dissertation, Universität Duisburg-Essen, Fakultät für Ingenieurwissenschaften, Institut für Mechatronik und Systemdynamik.

Mikelsons, L. & Su, Z. (2014). Simulation for Verification and Validation of Functional Safety. In: *Proceedings of the 10th International Modelica Conference, Lund, Sweden, March 2014*.

Mikelsons, L. & Worschech, N. (2012). A Toolchain for Real-Time Simulation using the OpenModelica Compiler. In: *Proceedings of the 9th International Modelica Conference, Munich, Germany, September 2012*.

Nelder, J. A. & Mead, R. (1965). A Simplex Method for Function Minimization. *The Computer Journal* **7**(4), 308–313.

Nilsson, B. (1989). *Structured Modelling of Chemical Processes: An Object-oriented Approach*. Ph.D. Thesis, Department of Automatic Control, Lund Institute of Technology (LTH).

Nissen, V. (1997). *Einführung in evolutionäre Algorithmen*. Vieweg+Teubner Verlag.

O'Neill, R. (1971). Algorithm AS 47: Function Minimization using a Simplex Procedure. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* **20**(3), 338–345.

Otter, M., Thuy, N., Bouskela, D., Buffoni, L., Elmqvist, H., Fritzson, P., Garro, A., Jardin, A., Olsson, H., Payelleville, M., Schamai, W., Thomas, E. & Tundis, A. (2015). Formal Requirements Modeling for Simulation-Based Verification. In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 2015*.

Pantelides, C. C. (1988). The Consistent Initialization of Differential-Algebraic Systems. *SIAM Journal on Scientific and Statistical Computing* **9**(2), 213–231.

Papageorgiou, M. (2012). *Optimierung: Statische, Dynamische, Stochastische Verfahren*. Springer Berlin Heidelberg.

Plail, M. (1998). *Die Entwicklung der optimalen Steuerungen: von den Anfängen bis zur eigenständigen Disziplin in der Mathematik*. Vandenhoeck & Ruprecht.

Reissig, G., Martinson, W. S. & Barton, P. I. (2000). Differential-Algebraic Equations of Index 1 May Have an Arbitrarily High Structural Index. *SIAM Journal on Scientific Computing* **21**(6), 1987–1990.

Reke, M. (2012). *Modellbasierte Entwicklung automobiler Steuerungssysteme in kleinen und mittelständischen Unternehmen*. Dissertation, Fakultät für Mathematik, Informatik und Naturwissenschaften, RWTH Aachen.

Runge, C. (1895). Über die numerische Auflösung von Differentialgleichungen. *Mathematische Annalen* **46**(2), 167–178.

Runge, T. F. (1977). *Universal Language for Continuous Network Simulation*. Ph.D. Thesis, Department of Computer Science, Illinois University, Urbana (USA).

Schamai, W., Fritzson, P., Paredis, C. & Pop, A. (2009). Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior Using Graphical Notations. In: *Proceedings of the 7th International Modelica Conference, Como, Italy, September 2009*.

Schmitz, R. (2007). Modelica. Vortrag innerhalb der AG Softwaretechnik des Lehrgebietes Software Engineering und Programmierung (Sommersemester 2007), Technische Universität Kaiserlautern.

Schramm, D. (2013). *Modellbildung und Simulation*. 2. Auflage, Klartext Medienwerkstatt GmbH, Essen, Universität Duisburg-Essen.

Seitz, M. (2015). *Speicherprogrammierbare Steuerungen für die Fabrik- und Prozessautomation: Strukturierte und objektorientierte SPS-Programmierung, Motion Control, Sicherheit, vertikale Integration*. Carl Hanser Verlag GmbH & Co KG.

Sendler, U. (2013). *Industrie 4.0 - Beherrschung der industriellen Komplexität mit SysLM*. Springer Vieweg.

Sjölund, M. (2015). *Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models*. Ph.D. Thesis, Department of Computer and Information Science, Linköping University.

Stein, B. & Husemeyer, U. (2001). Generierung heuristischer Modelle zur Diagnose. In: *Proceedings des 15. Symposium über Simulationstechnik (ASIM 01), Universität Paderborn, September 2001*.

Tarjan, R. (1972). Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing* **1**(2), 146–160.

Thiel, P. (2005). Gemischt-Ganzzahlige Nichtlineare Optimierung - Mixed-Integer Nonlinear Programming. Seminar über Optimierung bei Prof. Ulbrich und Prof. Gerdts, Juli 2005, Universität Hamburg.

Thieriot, H., Nemer, M., Torabzadeh-Tari, M., Fritzson, P., Singh, R. & Kocherry, J. (2011). Towards Design Optimization with OpenModelica Emphasizing Parameter Optimization with Genetic Algorithms. In: *Proceedings of 8th International Modelica Conference, Dresden, Germany, March 2011.*

Thomsen, T. (2003). MISRA C und seine Anwendbarkeit auf Seriencodegeneratoren. In: *Elektronik 25.*

Thümmel, M., Looye, G., Kurze, M., Otter, M. & Bals, J. (2005). Nonlinear Inverse Models for Control. In: *Proceedings of the 4th International Modelica Conference, Hamburg, Germany, March 2005.*

VDI2206 (2004). *VDI Richtlinie 2206: Entwicklungsmethodik für mechatronische Systeme.* VDI-Verlag, Düsseldorf.

VDI2221 (1993). *VDI Richtlinie 2221: Methodik zum Entwickeln und Konstruieren technischer Systeme und Produkte.* Beuth Verlag, Berlin.

VDI2422 (1994). *VDI Richtlinie 2422: Entwicklungsmethodik für Geräte mit Steuerung durch Mikroelektronik.* Beuth Verlag, Berlin.

VDMA (2015). *Trendstudie IT und Automatisierungstechnik.* Studie, durchgeführt von: VDMA.

VDW-Bericht (1997). *VDW: Abteilungsübergreifende Projektierung komplexer Maschinen und Anlagen.* Aachen, WZL.

Weigl, T. M. (2011). *Eine strukturbasierte Indexanalyse differential-algebraischer Gleichungen.* Dissertation, Technische Universität München, Lehrstuhl für Numerische Mathematik.

Wellstead, P. E. (1979). *Introduction to Physical System Modelling.* Academic Press London.

Wünsch, G. (2008). *Methoden für die virtuelle Inbetriebnahme automatisierter Produktionssysteme.* Forschungsberichte IWB, Band 215, Herbert Utz Verlag, München.

Yuan, Y.-x. (2008). Step-Sizes for the Gradient Method. *AMS IP Studies in Advanced Mathematics* **42**(2).

Zhao, X., Jin, Y., Ji, H., Geng, J., Liang, X. & Jin, R. (2013). An Improved Mixed-Integer Multi-Objective Particle Swarm Optimization and Its Application

in Antenna Array Design. In: *IEEE 5th International Symposium on Microwave, Antenna, Propagation and EMC Technologies for Wireless Communications (MAPE), Chengdu, China, October 2013*.

Zitzler, E., Laumanns, M. & Bleuler, S. (2004). A Tutorial on Evolutionary Multiobjective Optimization. In: *Metaheuristics For Multiobjective Optimisation*, Lecture Notes in Economics and Mathematical Systems. Springer Berlin Heidelberg, pp. 3–37.